2001

Formal verification in network of synchronizing FSMs with SPIN.

Fang. Li
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

FORMAL VERIFICATION IN NETWORK OF SYNCHRONIZING FSMS WITH SPIN

BY

FANG LI

A thesis
Submitted to the Faculty of Graduate Studies and Research
Through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada

May, 2001

Abstract

Concurrent systems are becoming more and more popular. Improving the qualities of these systems is an important issue we are facing. It is well known that developing concurrent software is a challenging task, mainly because of the non-determinism behavior involved in the system. One promising way to help the designer in this task is providing formal verification methods that can detect concurrency-related errors at the design stage. In this thesis, we present our study on model checking concurrency-related correctness of design artifacts for concurrent and distributed systems. We construct our model for design artifacts using a network of synchronizing finite state machines(NSFSM), which provides well known synchronization mechanisms from programming languages. The formal verification is performed by SPIN, which is based on visiting all the global system states reachable from a given initial state to check if some properties hold. The input language of SPIN is PROMELA, which features a C-like syntax, dynamic creation of processes, and various interprocess communication models. In order to apply SPIN, we provide an automatic translation from NSFSMs to PROMELA programs.

Keywords: concurrent system, non-determinism, design artifacts, formal verification, model checking, SPIN, PROMELA, synchronizing finite state machine.

Acknowledgements

The work presented in this thesis could not have been possible without the support of many people.

First, I would like to express my deepest appreciation to my supervisor, Dr. Xiaojun Chen, for giving me the opportunity to do research in formal verification study. Her work ethics, advice, enthusiasm, patience and encouragement has been an invaluable source of support and guidance that helped me finish my M. Sc. program.

I would also like to thank my committee members, Dr. Asish Mukhopadhyay, Dr. Kai Hildebrandt and Dr. Peter Tsin for spending their precious time to read this thesis and their comments, suggestion on this thesis work.

My special thanks go to the secretary of the School of Computer Science, Ms. Mary Mardegan, for her consistent helps. Thanks go to Xiubin, Xiaohong, Hanmei, Wayne, for their valuable advice, interesting discussions and encouragement.

I would like to thank my husband, Biao Zhou, for his understanding and support; and my parents, for their love and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1
INTRODUCTION

In this chapter, we give an introduction to the work of this thesis. It includes its aim, motivation and the thesis structure.

1.1 Aim

The aim of this thesis is to study and develop a new static analysis approach for checking the concurrency-related correctness of design artifacts for concurrent systems. This approach is based on a special kind of finite state machine (synchronizing finite state machine) and the existing model checker SPIN. After constructing the SFSM model of design artifacts, we provide automatic translation from the design of NSFSMs to PROMELA and check the correctness with SPIN.

1.2 Motivation

Concurrent and distributed systems are much more complex to analyze statically or test dynamically due to the nondeterminism involved. Although some reproducible testing techniques have been developed, it is still difficult to discover concurrency-related errors. Static analysis approaches increase our confidence in the correctness of concurrent and distributed systems. This thesis presents our study on statically analyzing concurrency-related correctness of design artifacts for multi-process systems. We know that it is usually less expensive to correct errors in the design phase than to correct them at the coding stage. Our main concern is that any element in a description language for design documents could be based on existing concepts well known for ordinary designers and programmers.

Model checking is a formal verification technique which checks the consistency between a requirement specification and a behavior model of the system by exploring the state space of the model. The model of the abstract behavior of a multi-process (here process

means either threads in the same process or processes in distributed systems) can be usually decomposed into the description of the behavior of each process and the description of the interactions among these processes. Here we assume that the behavior of each process is described in terms of a Synchronizing Finite State Machine (SFSM), a special kind of Finite State Machine (FSM), where transitions in the state machines may contain information for synchronization. Such information is expressed by way of some well-known synchronization mechanism such as semaphore and monitor.

We have chosen FSM in our study because it is an effective tool to model system behavior at an abstract level, and has been widely accepted and used in many application domains such as sequential circuits, lexical analysis, pattern matching, communication protocols. Recently it has also been included into Unified Modeling Language (UML), a well known set of graphical design notations, as a means to describe the behavior of a system, of a subsystem, of an object, etc., on various levels of abstraction.

We have chosen semaphore and monitor synchronization mechanism to be used at the design stage mainly based on the following observations:

- Semaphore and monitors are well known concepts for ordinary designers and well-mastered by many programmers. Thus, adopting these concepts at the design stage is obviously the easiest way to prevent us from adding burden to both designers and programmers to learn and use new concepts and design notations.
- Most of the discussions on modeling concurrent systems, either via synchronous mode or via asynchronous mode, require process identification for the process communications. That is, communications are among processes that are known a priori. This may enlarge the distance between design artifacts and the final code. For example, in the coding stage, when we use method *notify* defined in Java, we do not know in reality which process will be notified. As a direct consequence of this distance, the verification of the correctness of the final code (or a static analysis model of it) against the design specification may sometimes turn out to be more difficult.

In general, the communications among multiple processes can be realized either via procedural/method calls or via shared objects. The coordination of the access to a procedure/method or to a shared object can be realized by means of semaphores, monitors etc. In this thesis, we consider synchronization among multiple processes via method calls or shared objects, governed by semaphores and monitors. This is based on various reasons. First, the concepts of semaphores and monitors have been well accepted by both the academic and industrial communities as sound mechanisms for concurrency control. Second, the definition of these mechanisms is at an abstract level. We can concentrate on concurrency-related controls of a system, ignoring other details.

A network of SFSMs is a set of SFSMs together with an environment shared by the SFSMs. Given a network of SFSMs, we provide an operational semantics in terms of labeled transition systems, which describes the global behavior of the system. The defined formal model is the basis for formal reasoning about the correctness of a design against certain properties that, due to the nondeterminism involved, may be hard to detect by testing the final code.

However elegant an analysis approach is, if it cannot be automated, it will not be practical for verification of large-scale concurrent systems. There exist some automatic model checking tools [4, 7, 11, 18, 23, 28]. SPIN is one of the well studied and used model checkers that explores the enumerative state space. The main design goal of SPIN is to verify and validate asynchronous process protocols, thus it is suitable to verify our NSFSM design notation. The input language of SPIN is PROMELA, which features a C-like syntax, dynamic creation of processes and various interprocess communication models. So we need a bridge between the NSFSM design notation and the model checker SPIN. We provide a translator which can automatically translate the design of NSFSM into PROMELA program. The translator is implemented and is running properly.

1.3 Thesis Structure

This thesis consists of ten chapters. Chapter 2 discusses the features of concurrent systems and the semaphore and monitor synchronization mechanism. Chapter 3 gives an overview of finite state based verification techniques. Chapter 4 describes the basic concept of a finite state machine, the synchronizing finite state machine (SFSM) and the network of SFSM (NSFSM). Chapter 5 introduces the model checker SPIN and its input language PROMELA. Chapter 6 describes our translation rules from network of SFSMs to PROMELA. In chapter 7, we give implementation details of the translator. In chapter 8, we present the evaluation. Chapter 9 shows some related work and the advantages of our approach. Chapter 10 indicates the conclusions and future work.

Chapter 2

CONCURRENT SYSTEMS AND SYNCHRONIZATION MECHANISMS

2.1 Features of Concurrent System

An 'ordinary' software system consists of sequential instructions. A concurrent system is a set of ordinary sequential program executed in abstract parallelism (concurrently) [2]. Each sequential program can be considered as a process.

Concurrent system abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes. We will consider possible interaction in two cases:

- **Contention** Two processes compete for the same resource: computing resources in general, or access to a particular memory cell or channel in particular.
- **Communication** Two processes may need to communicate causing information to be passed from one to the other. Even the mere fact of communication can be important because it allows the processes to synchronize: to agree that a certain event has taken place

A concurrent system is required to be correct under all interleavings. Then, if the computer hardware is changed or if the rate of incoming signals changes, we may have a different interleaving, but the system is still correct. Furthermore, any concurrent system that depends on external signals is difficult, if not impossible, to repeat exactly. A concurrent system can not be 'debugged' in the familiar sense of diagnosing a problem, correcting it and rerunning the program to check if the bug still exists. We may just execute a different interleaving in which the bug does not exist. The solution is to develop reproducible testing and verification techniques that ensure a concurrent system is correct under all interleavings. The only constraint to arbitrary interleaving is that

fairness must be preserved which means that no process in a shared system is deferred forever.

Correctness Requirement

Safety properties: The property must always be true.

The most common safety property is mutual exclusion: two processes may not interleave certain (sub-) sequences of instructions. Instead, one sequence must be completed before the other commences. The order in which the sequences are executed is not important.

The other important safety property is absence of deadlock. A non-terminating system must always be able to proceed doing useful work. A system which cannot respond to any signal or request is deadlocked and again, this must never happen. A system may be quiescent if there is nothing to do, but in that case, it is usually actively executing a background idle-process or self-test and it will be able to respond instantly to an interrupt.

Liveness properties: the property must eventually be true ('now' is included in 'eventually').

A particular type of liveness property is called fariness property. If there is contention, we will often want to specify how the contention should be resolved. Four possible specifications of fairness are:

Weak fairness If a process continuously makes a request, eventually it will be granted.
Strong fairness If a process makes a request infinitely often, eventually it will be granted.
Linear waiting If a process makes a request, it will be granted before any other process is granted the request more than once.
FIFO (first-in-first-out) If a process makes a request, it will be granted before that of any process making a later request.

2.2 Semaphore and Monitor

2.2.1 Introduction to Semaphore

A semaphore is an integer-valued variable which can take only non-negative values. Exactly two operations are defined on a semaphore S:

Wait(S) If S > 0 then S:= S - 1 else suspend the execution of this process. The process is said to be suspended on the semaphore S.
Signal(S) If there are processes that have been suspended on this semaphore, wake one of them else S:= S+1.

The semaphore has the following properties:

1. **Wait(S)** and **Signal(S)** are atomic instructions. In particular, no instructions can be interleaved between the test that S>0 and the decrement of S or the suspension of the calling process.
2. A semaphore must be given a non-negative initial value.
3. The **Signal(S)** operation must wake one of the suspended processes. The definition does not specify which process will be awakened.

A semaphore that can take any non-negative value is called a **general semaphore** [2]. A semaphore which takes only the values 0 and 1 is called a **binary semaphore** [2] in which case **Signal(S)** is defined by: if...else S:= 1.

There are many definitions of semaphores in the literature. It is important to be able to distinguish between the various definitions because the correctness of a program will depend on the exact definition used. In this thesis, we will use busy-wait semaphore definition.

Busy-wait semaphore The value of S is tested in a busy-wait loop. The entire if-statement is executed as an atomic operation, but there may be interleaving between cycles of the loop.

- **Wait(S):**

  Loop

  If S > 0 then S := S - 1; exit; end if;

  End loop;
- **Signal(S):** S := S + 1.

2.2.2 Introduction to Monitor

Using semaphores, we can give solutions to common concurrent programming problems. However, the semaphore is still a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores is diffused among all the implementers of the system. If one of them forgets to call **Signal(S)** after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.

Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into a few modules. Monitors are a generalization of the monolithic monitor found in operating systems. Critical sections such as allocation of I/O devices and memory, queuing requests for I/O, and so on, are centralized in a privileged program. Ordinary programs request services that are performed by the central monitor. These programs are run in a hardware mode that ensures that they cannot be interfered with by ordinary programs. Because of the separation between the system and its applications programs, it is usually clear who is at fault if the system crashes (though it may be extremely difficult to diagnose the exact reason).

The monitors discussed in this thesis are decentralized versions of the monolithic monitor. Rather than having one system program handle all requests for services

involving shared devices or data structures, we define a separate monitor for each object or related group of objects. Processes request services from the various monitors. If the same monitor is called by two processes, the implementation ensures that these are processed serially to preserve mutual exclusion. If different monitors are called, their executions can be interleaved. The syntax of monitors is based on encapsulating items of data and the procedures that operate upon them in a single module. The interface of a monitor consists of a set of procedures (and/or methods). These procedures operate on data that are hidden within the module. The difference between a monitor and an ordinary module (or class) such as an Ada package is that a monitor not only protects internal data from unrestricted access but also synchronizes calls to the interface procedures (or methods). The implementation ensures that the procedures are executed under mutual exclusion. In this thesis, we use the synchronization primitive that will allow a process to suspend itself if necessary and other processes can resume the suspended process.

The mutual exclusion requirement is satisfied by the definition of monitor, in which only one process is allowed to execute a monitor procedure at any time. For synchronization, a structure called condition variable is defined. A condition variable C has three operations defined upon it.

- **M_Wait(C)** The process that called the monitor procedure containing this statement is suspended on a FIFO queue associated with C. The mutual exclusion on the monitor is released.

- **M_Signal (C)** If the queue for C is non-empty then wake the process at the head of the queue.

- **M_Non-Empty (C)** A boolean function that returns true if the queue for C is non-empty.

The **M_Wait** operation allows a process to suspend itself and releases the mutual exclusion on the monitor. The **M_Signal** operation will be ignored if no process is suspended on the queue for C.

2.2.3 Combination of Semaphore and Monitor

Based on the above definitions of semaphore and monitor, we propose a new concept: resource-condition pair (RC-Pair), which is a combination of busy-wait semaphore and monitor.

We know that semaphore mechanism ensures mutual exclusion. The initial value of a semaphore indicates the number of resources that will be mutually exclusively accessed. For example, in a system there are three printers, but there are five processes who need to print their files. If we use semaphore to control the printers, the initial value of semaphore will be three. Figure 2.1 is a solution to this problem. At most three processes are printing because after three times execution of Wait(S), the value of S will be zero and other processes who need to print will have to wait until some process finishes printing and executes Signal(S).

```
S: Semaphore := 3;

Task body Pi (i := 1...5) is
Begin
        Loop
            Non_Critical_Section_1;
            Wait(S);
             Print;
             Signal(S);
        End loop;
End Pi;
```

Fig. 2.1 Mutual exclusion with semaphores

Different from semaphore mechanism, monitor mechanism ensures both mutual exclusion and synchronization among processes. A process will check some condition after it locks a monitor. If the condition is not satisfied, the process will suspend itself and put itself in the waiting queue of that condition in addition to release the monitor. The suspended process can only be resumed when some other process that locks the monitor and calls Signal(Condition). Let us take a look at the monitor in producer-consumer problem:

- The only operations permitted on a bounded buffer are produce and consume an item.
- Produce and consume exclude each other.
- A producer will suspend on a full buffer and a consumer on an empty buffer

```
monitor Producer_Consumer_Monitor is
    B: array(0..N-1) of Integer;
    In_Ptr, Out_Ptr: Integer := 0;
    Count: Integer := 0;
    Not_Full, Not_Empty: Condition;

Procedure Produce(I: in Integer) is
Begin
    If Count = N then M_Wait(Not_Full); end if;
    B(In_Ptr) := I;
    In_Ptr := (In_Ptr + 1) mod N;
    M_Signal(Not_Empty);
End Produce;

Procedure Consume(I: out Integer) is
Begin
    If Count = 0 then M_Wait(Not_Empty); end if;
    I := B(Out_Ptr);
    Out_Ptr := (Out_Ptr + 1) mod N;
```

```
    M_Signal(Not_Full);
End Consume;
```

Fig. 2.2 Monitor for producer-consumer

Now we combine the two mechanisms and define a new concept RC_Pair which contains both a number of resources and some waiting conditions and can fulfill both mutual exclusion and synchronization control. RC_Pair contains two parameters: numResource and numCondition.

If numResource > 0 and numCondition == 0, the RC_Pair is a semaphore;
If numResource ==1 and numCondition > 0, the RC_Pair is a monitor;

RC_Pair is a general form for concurrency control.

We define five operations on RC-Pair:

Lock: The processes compete for a certain resource, the winner will lock the resource and the loser will keep checking the availability of the resource.

Unlock: The process that locks the resource finishes its task and releases the resource.

Suspend: A process who has locked the RC_Pair and decides to wait on some condition. It automatically calls Unlock.

Resume: A process locks the RC_Pair and wakes up the first process in some condition's waiting queue.

ResumeAll: A process locks the RC_Pair and wakes up all the processes in some condition's waiting queue.

Chapter 3

FINITE-STATE VERIFICATION TECHNIQUES

Automated or semi-automated techniques for checking the correctness of concurrent programs range from formal verification that the program satisfies a complete specification to dynamic methods that examine the outcome of executions of the program or check assertions during such executions [1]. Different techniques are, of course, useful for different purposes. For the most critical modules, for example, the extra assurance of formal verification may justify its effort and expenses, but formal verification is likely to be impractical if not infeasible for large and complex programs. Dynamic techniques such as testing, on the other hand, while an essential part of software development, examine only a single execution at a time. Especially for concurrent programs, which may display very different behaviors in response to the same input data due to changes in the relative order of events in different components, such techniques may fail to detect serious faults.

A third class of techniques consists of finite-state verification methods such as model checking, necessary conditions analysis, or data flow analysis. Such techniques can consider all possible executions of a concurrent program, but generally cannot be used to show that a program satisfies a complete specification. Instead they check a particular property or collection of properties, such as freedom from deadlock or the mutually exclusive use of certain resources, that should hold on all executions of the program. Because these methods can be automated relatively easily, at least in comparison to the theorem-proving required for formal verification, but still consider all possible executions of the program, they have the potential to play an extremely important role in the development of high-quality concurrent software. With the spread of distributed applications running over the Internet and the widespread adoption of languages such as Java that provide support for concurrency, the need for developers of concurrent programs to use such methods will continue to increase.

A variety of finite-state verification techniques have been proposed, and several of these have been implemented as prototype tools. In this thesis, we will concentrate on finite-state machine model checking.

A Model is a simplified representation of the real world [22] and, as such, includes only those aspects of the real-world system relevant to the problem at hand. For example, a model car, used in speed tests, models only the external shape and the engine of the car. The color of the car will not affect the car's speed properties. Models are widely used in engineering since they can be used to focus on a particular aspect of a real-world system such as the speed of a car. Our models represent the behavior of real concurrent systems. The models abstract much of the detail of real programs concerned with data representation, resource allocation and user interaction. They let us focus on concurrency. We can animate these models to investigate the concurrent behavior of the intended program. More importantly, we can *mechanically* verify that a model satisfies particular safety and progress properties, which are required of the program when it is implemented. This formal verification is made by a model-checking tool *SPIN.* Exhaustive model checking using SPIN allows us to check for both desirable and undesirable properties for all possible sequences of events and actions. Our models are based on finite state machines. Finite state machines are familiar to many programmers and engineers. They are used to specify the dynamic behavior of objects in well-known object -oriented design methods. For those not yet familiar with state machines, they have an intuitive easily grasped semantics and a simple graphical representation. The state machines used in this thesis is synchronizing finite state machine (SFSM), which facilitate formal analysis and mechanical checking, thus avoiding the tedium and error introduction inherent in manual formal methods. The details of our finite state machine model and SPIN model checker are described in later chapters.

Chapter 4

NETWORK OF SYNCHRONIZING FINITE STATE MACHINE

At a low level of abstraction, a design artifact is most easily understood as a state machine. Design criteria can also easily be expressed in terms of desirable or undesirable states and state transitions. In a way, the protocol state symbolizes the assumptions that each process in the system makes about the others. It defines what actions a process is allowed to take, which events it expects to happen, and how it will respond to those events. We will concentrate on formal validation of concurrent system design. There exist many variations of the basic finite state machine model. Rather than list them all, we will introduce a special finite state machine: synchronizing finite state machine.

4.1 Definition of Basic Finite State Machine

A finite state machine is usually specified in the form of a transition table [17], much like the one shown in Table 1 below.

Table 4.1 - Mealy

Condition		Effect	
Current State	In	Out	Next State
Q0	-	1	Q2
Q1	-	0	Q0
Q2	0	0	Q3
Q2	1	0	Q1
Q3	0	0	Q0
Q3	1	0	Q1

For each control state of the machine the table specifies a set of transition rules. There is one rule per row in the table, and usually more than one rule per state. The example table contains transition rules for control states named q0, q1, q2, and q3. Each transition rule has four parts, each part corresponding to one of the four columns in the table. The first two are conditions that must be satisfied for the transition rule to be executable. They specify

- The control state in which the machine must be
- A condition on the "environment" of the machine, such as the value of an input signal

The last two columns of the table define the effect of the application of a transition rule. They specify

- How the "environment" of the machine is changed, e.g., how the value of an output signal changes
- The new state that the machine reaches if the transition rule is applied

In the traditional finite state machine model, the environment of the machine consists of two finite and disjoint sets of signals: input signals and output signals. Each signal has an arbitrary, but finite, range of possible values. The condition that must be satisfied for the transition rule to be executable is then phrased as a condition on the value of each input signal, and the effect of the transition can be a change of the values of the output signals. The machine in Table 1 illustrates that model. It has one input signal, named In, and one output signal, named Out.

A dash in one of the first two columns is used as a shorthand to indicate a "don't care" condition (that always evaluates to the boolean value *true*). A transition rule, then, with a dash in the first column applies to all states of the machine, and a transition rule with a dash in the second column applies to all possible values of the input signal. Dashes in the last two columns can be used to indicate that the execution of a transition rule does not change the environment. A dash in the third column means that the output signal does not change, and similarly, a dash in the fourth column means that the control state remains unaffected.

In each particular state of the machine there can be zero or more transition rules that are executable. If no transition rule is executable, the machine is said to be in an *end-state.* If precisely one transition rule is executable, the machine makes a deterministic move to a new control state. If more than one transition rule is executable a nondeterministic choice is made to select a transition rule. A nondeterministic choice in this context means that the selection criterion is undefined. Without further information either option is to be considered equally likely. From here on, we will call machines that can make such choices nondeterministic machines. Table 2 illustrates the concept. Two transition rules are defined for control state **q1.** If the input signal is one, only the first rule is executable. If the input signal is zero, however, both rules will be executable and the machine will move either to state **q0** or to state **q3**.

Table 4.2 -- Non-Determinism

Concurrent State	In	Out	Next State
Q1	-	0	Q0
Q1	0	0	Q3

The behavior of the machine in Table 1 is more easily understood when represented graphically in the form of a state transition diagram, as shown in Figure 1.

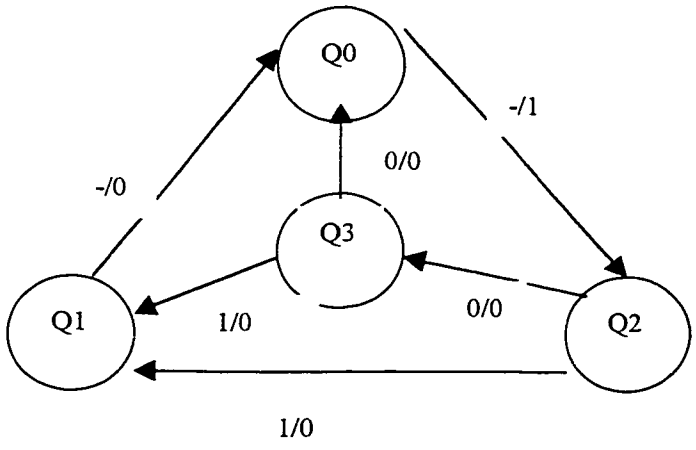


Fig. 4.1 -- State Transition Diagram

The control state are represented by circles, and the transition rules are specified as directed edges. The edge labels are of the type *c/e*, where *c* specifies the transition condition (e.g. the required set of input values) and *e* the corresponding effect (e.g. a new assignment to the set of output values).

4.2 Synchronizing Finite State Machine

We use SFSM [6] to describe the abstract behavior of a process. A SFSM is a finite state machine whose transitions may contain information about its synchronization with the environment expressed by some typical synchronization mechanisms such as monitor. In this thesis, we consider the synchronization mechanisms from RC_Pair.

As we described in Chapter 2, RC_Pair is a combination of semaphore and monitor. The synchronization mechanism of RC_Pair comes from monitor, but they have some differences. The operation of M_Wait(C) in monitor is decomposed into two operations in RC_Pair: unlock and suspend(C) (of course, the precondition of M_Wait(C) is that it has locked a RC_Pair). When RC_Pair is displayed as a monitor, a process can lock the RC_Pair mutual exclusively, then check if the waiting condition C is satisfied. If the process needs to wait for C, it suspends itself on C. Signal(C) in monitor can be represented in RC_Pair as resume(C) or resumeAll(C). We introduce the resumeAll operation because in design stage we do not know what kind of implementation language will be used in coding. Both Java and Ada have built-in concurrency support, the monitor mechanism. If we use a Java monitor, the parameter numResource of RC_Pair should be 1 and the parameter numCondition of RC_Pair should be 1, too. Java contains a statement *notifyAll( )* which will wake up all the processes in the waiting queue. In design notation, we define a corresponding operation resumeAll(C) which can be implemented directly with notifyAll( ) if we choose Java as the implementation. Please note that, whether resume(C) or resumeAll(C), we just wake up the process(es) in the waiting queue without any guarantee that the awaked process will get the resource. They must compete for the resource again. Here, we do not consider the immediate resumption requirement in Ada monitor mechanism.

We describe the behavior of a process by means of all possible transitions of the process from one state to another caused by actions. Each action is associated with

- A set of preconditions on the environment, called *guard*, which must be satisfied in order to perform the action, the **lock** operation of RC_Pair may be among the preconditions;
- A set of postconditions on the environment right after this action, which may include the **unlock** operation of RC_Pair.

Thus, intuitively, transition $(s,(g,(a),ass),s')$ means that (I) the process currently in state s is capable of doing action a ending in state s', if guard g is satisfied by the environment; (II) If guard *g* contains any *lock* operation of RC_Pair, it means action *a* need to occupy some resource exclusively; and (III) action *a* must be followed by the assignments expressed by *ass*, including the *unlock* operation of some RC_Pair. Without loss of generality, we assume that *suspend(C)*, *resume(C)* and *resumeAll(C)* are special actions.

Formally, a SFSM is a 6-tuple (S, Act, V, RC, $\rightarrow$, s^0) where

1. S is a finite set of states;
2. Act is a finite set of actions including suspend, resume and resumeAll
3. V is a set of variables;
4. RC is a set of RC_Pairs;
5. $\rightarrow$ is a transition relation $\rightarrow \subseteq S \times (Guard_V \times (RC \cup \{-\} \times Act) \times PostAss_V) \times S$;
6. $s^0 \in S$ is the initial state.

Here $Guard_V$ denotes a set of boolean expressions with variables in V and *lock* operations, and $PostAss_V$ a set of assignments to variables in V with the evaluation from

the expressions of variables in V and *unlock* operation. In the following, we also use $s \xrightarrow{g,(a),ass} s'$ for $(s,(g,(a),ass),s') \in \rightarrow$.

A SFSM can be represented graphically by a directed graph (V, E) with a special arrow pointing to the initial state. Set V of vertices represents the set of states of the process, and set E of edges represents all specified transitions. Each edge $(v_i, v_j; (g,(a),ass)) \in E$ is a transition from state v_i to v_j with guard g, action a, and post assignments ass.

4.3 Network of Synchronizing FSMs

Given a concurrent or distributed system with the behavior of each of its process described in a SFSM, the behavior of the global system is expressed by the network of these SFSMs, i.e. the set of SFSMs communicating with each other via a shared environment(Figure 4.1). The formal model for a network of SFSMs is constructed in two steps. In the first step, we make transition reification to all those *suspend* transitions, i.e. transition $(g,(suspend),-)$ for a guard g which must be in a state that some RC_Pair is locked. In the second step, the operational semantics based on the SFSMs from the first step is defined in terms of labeled transition system by way of a set of structural rules.

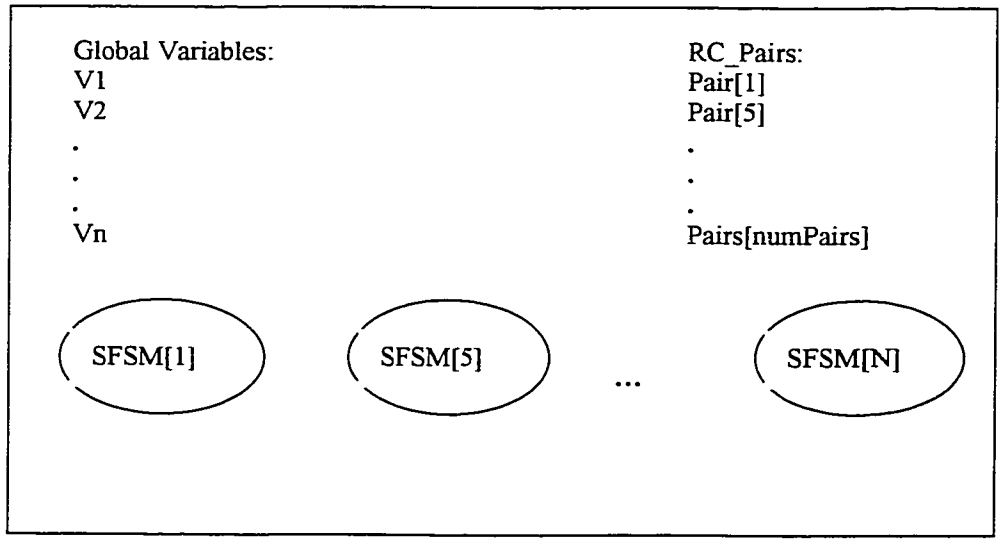


Fig. 4.2 Network of SFSMs and shared environment

4.3.1 Transition Reification

According to the meaning of operation *suspend(C)* defined for RC_Pair, we know that the *suspend* action used in SFSM can be further decomposed into two continuous atomic actions: one is to put the process itself to the waiting queue of the corresponding condition C, and the other is to be resumed, ready to execute the next activity. We reify all transitions related to *suspend* activity into two activities according to this interpretation.

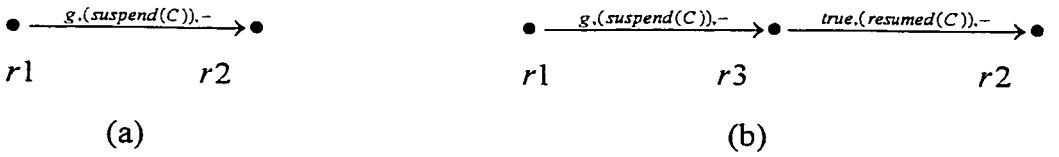


Fig.4.3: Transition reification: (a) the original one; (b) the transformed

Figure 1 illustrates the reification: Part (a) shows the original transition from state r1 to state r2, with guard g which are in locking state of some RC_Pair. Part (b) shows the resulting reified transitions constituted by the transition from state r1 to a newly introduced state r3, and the transition from r3 to state r2. The first transition has guard g. The second has no guard, but actually has released the locked RC_Pair. Here *resumed* is a special action that will be used to synchronize with action *resume* performed by another process.

4.3.2 Operational Semantics

A network of SFSMs consists of a set of SFSMs and an environment shared by all these SFSMs. Let $ST_i = (S_i, Act_i, Var_i, RC_i, \rightarrow_i, s_i^0)$ for $1 \le i \le n$ be n SFSMs. A state in the network of $ST_1, \cdots, ST_n$ consists of a state in each of the SFSM ST_i together with an environment. Typically, such an environment contains information about the status of the shared objects and RC_Pairs, i.e.

- the values of each variable in V where $V = \bigcup_{1 \le i \le n} Var_i$;

- the states of the waiting queues for each condition of RC_Pair in RC where $RC = \bigcup_{1 \le i \le n} RC_i$;

We use $Env_{V,RC}$ to denote such an environment. Moreover, for $env \in Env_{V,RC}$, we will use $env(Q_{rc(C)})$, where $rc \in RC$ and C is a waiting condition of rc, to denote the waiting queue of condition C of rc in env. The elements in $env(Q_{rc(C)})$ are integers, i.e. the process id denoting the corresponding processes that are currently waiting in the queue of $rc(C)$. In addition, let q be a waiting queue. We will use

- ϕ to denote an empty queue;
- $top(q)$ to denote the first element in q;
- $enqueue(q,i)$ to denote the waiting queue obtained from q by adding element i to its tail;
- $dequeue(q)$ to denote the waiting queue obtained from q by removing the first element;
- $i \in q$ to denote that i is an element in q.

Furthermore, let $x_i \in V$, e_i be expressions on V (for $1 \le i \le n$). We use $env[x_1 / e_1,...,x_k / e_k]$ to denote the environment obtained from env by substituting the evaluations of x_i with the evaluation of e_i. Similarly, we use $env[Q_{rc(C)} / x]$ to denote the environment obtained from env by substituting the states of $Q_{rc(C)}$ with the evaluation of x.

Recall that in SFSMs, the information carried on each transition consists of an action together with a guard, and a set of assignments to be performed after the action. For the network of SFSMs, we add one more piece of information in addition to the above to each transition, that is, the identifications of the processes who have participated in this system evolution.

The operational semantics of a network of SFSMs is defined in terms of labeled transition systems. In general, a labeled transition system is a quadruple $(S, L, \rightarrow, s^0)$ where

- S is a set of meaningful intermediate states during the execution of the system.
- L is a set of labels about the activity. Such a label carries all related information that we concern about this action from the external world.
- $\rightarrow \subseteq S \times L \times S$ is a transition relation describing how the computation proceeds: $s \xrightarrow{l} s'$ means that the system in state s may perform a transition into state s' with label l.
- s^0 is the initial state of the system.

According to our above discussion, the labeled transition system for the network of $ST_1, \cdots, ST_n$ is $(S, L, \rightarrow, s^0)$ where

- $S = Env_{V.RC} \times S_1 \times \cdots \times S_n$.
- $L = P(N) \times Guard_V \times (\bigcup_{1 \le i \le n} Act_i) \times PostAss)$ where N denotes the set of natural numbers. $P(N)$ denotes the powerset of N, and its element denotes a set of indices of the SFSMs, i.e. the identifications of the processes who have participated in this action.
- $\rightarrow \in S \times L \times S$ is the transition relation. We will give more details below on how it is defined.
- $s^0 = \langle env_{init}, s_1^0, \cdots, s_n^0 \rangle$ is the initial state.

The transition relation $\_ \xrightarrow{-} \_$ is defined as the least relation satisfying the following five *structural rules*. All the structural rules have the following schema:

$$\frac{\text{ANTECEDENT}}{\text{CONSEQUENT}} \text{SIDE} - \text{CONDITION}$$

which is interpreted logically as:

$$\forall(\text{ANTECEDENT} \wedge \text{SIDE} - \text{CONDITION} \rightarrow \text{CONSEQUENT})$$

where $\forall(\cdots)$ stands for the universal closure of all free variables occurring in $(\cdots)$. In case either the ANTECEDENT or SIDE – CONDITION is missing, they are interpreted as true.

In the following, $s_i, s_i' \in S_i, s_j, s_j' \in S_j$ for $1 \le i, j \le n, i \ne j, g \in Guard_{Var_i}, rc \in RC$, $ass \in PostAss_{Var_i}, env \in Env_{V,RC}, s \in S_1 \times \cdots \times S_n$. For $1 \le u \le r$, $1 \le r \le n$, $1 \le i_u \le n, s_{i_u}, s_{i_u}' \in S_{i_u}$, and we use $s(s_{i_1}, \cdots, s_{i_r})$ to denote state s where the states in SFSM ST_{i_u} is s_{i_u}, and we use $s(s_{i_1}/s_{i_1}', \cdots, s_{i_r}/s_{i_r}')$ to denote state s where SFSM state s_{i_u} is substituted by s_{i_u}'. Moreover, we use $env \perp g$ to denote that in environment env, the evaluation of guard g returns *true*.

Rule (a)

$$\frac{s_i \xrightarrow{g,(a),ass}_i s_i'}{\langle env, s(s_i) \rangle \xrightarrow{\{i\},g,(a)ass} \langle env[ass], s[s_i/s_i'] \rangle} env \perp g$$

for any $a \in Act - \{suspend, resume, resumeAll, resumed\}$. Here

$$env[ass] = \begin{cases} \text{env} & \text{if ass} = \phi \\ \text{env}[x_j / \exp_j][ass'] & \text{if ass}=\{x_j = \exp_j\} \cup ass' \end{cases}$$

The first rule says that if process i can evolve from state s_i to state s_i' by performing action a under condition g and with post assignments ass, and g is satisfied in the current environment env, then the whole system with process i in state s_i can evolve into another state where process i in state s_i', all other processes remains in the same state, and in the new environment, some variables have change their values according to ass. Note that a can be any action except for *suspend, resume, resumeAll and resumed.* The system evolutions caused by these special actions are given by the other rules.

Rule (b)

$$\frac{s_i \xrightarrow{g,(suspend(C)),\phi}_i s_i'}{\langle env, s(s_i)\rangle \xrightarrow{\{i\},g,(suspend(C)),\phi} \langle env[Q_{rc(C)} / enqueue(Q_{rc(C)}, i)], s[s_i / s_i']\rangle} env \perp g$$

Rule (b) is to handle the evolution of the system caused by special *suspend* action. This rule is similar to Rule (a) except that in the ending state of the system, the waiting queue of condition C of rc is enqueued by process i.

Rule (c)

$$\frac{s_i \xrightarrow{g,(a),\phi}_i s_i'}{\langle env, s(s_i)\rangle \xrightarrow{\{i\},g,(a),\phi} \langle env, s[s_i / s_i']\rangle} env \perp g, env(Q_{rc(C)}) = \phi$$

for any a = *resume* or *resumeAll.*

Rule (c) is used to define the possible evolution of the system caused by the *resume* or *resumeAll* action of process i when there is no process in the waiting queue of the corresponding condition C of rc. In this case, the whole system makes the related transition corresponding to the one by process i while all other processes remain in the same state.

Rule (d)

$$\frac{s_i \xrightarrow{g,(resume(C)),\phi}_i s_i' \wedge s_j \xrightarrow{-,(resumed(C)),\phi}_j s_j'}{\langle env, s(s_i, s_j)\rangle \xrightarrow{\{i,j\},g,(resume(C)),\phi} \langle env[Q_{rc(C)} / dequeue(Q_{rc(C)})], s[s_i / s_i', s_j / s_j']\rangle}$$

$$env \perp g, top(env(Q_{rc(C)})) = j$$

Rule (d) is used to define the possible evolution of the system caused by the *resume* action of process i when the waiting queue of the corresponding condition C of rc is nonempty. In this case, the whole system makes the related transition corresponding to the simultaneous transitions by process i with action *resume* and by process j, which is the head in the waiting queue, with action *resumed.*

Rule (e)

$$\frac{s_i \xrightarrow{g.(resumeAll(C)).\phi}_i s_i' \wedge \Lambda_{1\le u\le r} s_{j_u} \xrightarrow{-.(resumed(C)).\phi}_{j_u} s_{j_u}'}{\langle env, s(s_i, s_{j_1}, \cdots, s_{j_r})\rangle \xrightarrow{\{i.j_1.\cdots j_r\}.g.(resume(C)).\phi} \langle env[Q_{rc(C)}/\phi], s[s_i/s_i', s_{j_1}/s_{j_1}', \cdots, s_{j_r}/s_{j_r}']\rangle}$$

$$env \perp g, \forall l \in env(Q_{rc(C)}), \exists p \quad s.t. \qquad j_p = l \text{ for any } 1 \le r \le n.$$

Rule (e) is used to define the possible evolution of the system caused by the *resumeAll* action of process i when the waiting queue of the corresponding condition C of rc is nonempty. In this case, the whole system makes the related transition corresponding to the simultaneous transitions by process i with action *resumeAll* and by processes $j_1 \cdots j_r$, which are those and only those processes in the waiting queue, with action *resumed*. Here "processes $j_1, \cdots j_r$ are those and only those threads in the waiting queue" is expressed by the following two conditions:

$$s_{j_u} \xrightarrow{-.(resumed).\phi}_{j_u} s_{j_u}'$$

in the sense that process $j_u (1 \le u \le r)$ can evolve from state s_{j_u} to state s_{j_u}' by action resumed on condition C of rc. This implies that process j_u is in the waiting queue of condition C of rc.

$\forall l \in env(Q_{rc(C)}), \exists p \quad s.t. \quad j_p = l$

in the sense that all processes in the waiting queue of condition C of rc have participated in the synchronized transition.

Given an initial configuration $(env_{init}, \{s_1^0, \cdots, s_n^0\})$, the above structural rules allow us to associate to it a labeled transition system whose states are those reachable from $(env_{init}, \{s_1^0, \cdots, s_n^0\})$, via the transitions inferred by using the structural rules. The labeled transition system generated by the initial states of each SFSM describes all the possible evolutions, and hence constitutes our model of the system.

Chapter 5
PROMELA AND SPIN

5.1 Overview of SPIN

SPIN [17] [18] [20] [29] is a generic verification system that supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these.

As a formal methods tool, SPIN aims to provide:

1) an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,
2) a powerful, concise notation for expressing general correctness requirements, and
3) a methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

The analysis engine of SPIN implements the technique known as reachability analysis, which is based on visiting all the global system states reachable from a given initial state to check if some properties hold. This technique is powered in SPIN with a partial order reduction aimed at restricting the state path to be visited.

SPIN accepts design specifications written in the verification language PROMELA and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic(Figure 5.1).

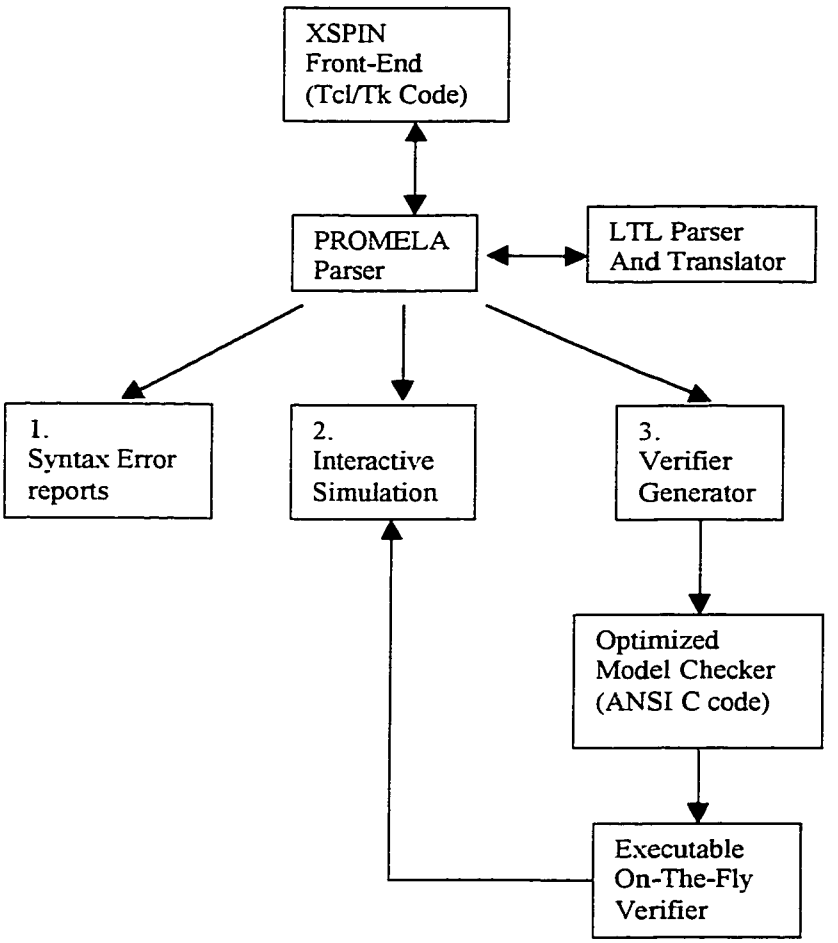


Figure 5.1 The structure of SPIN simulation and verification

5.2 Modeling Language PROMELA

PROMELA features a C-like syntax, dynamic creation of processes, and various interprocess communication models.

The PROMELA language provides both the elements aimed at describing a set of extended communicating finite state machines and those for describing the claims that must be verified. The latter can be expressed for example as state properties, i.e. predicates that must hold in specific process states, or temporal claims, specifying temporal orderings of state properties. The latter may take the form of temporal logic formulas. The description of a system takes a form similar to a C program file: it is a

collection of proctype declarations, each of which defines the behavior of a process and takes a syntactical form similar to the one of C functions:

```
Proctype <identifier> (<formal par. List>) {
        <statement list>
}
```

A PROMELA file is pre-processed by the standard C preprocessor, i.e. it may contain directives such as #define or #include, and C-style comments. Variables can be declared and used as in the C language. Variables declared inside a proctype definition are private variables of each instance of the proctype, whereas those defined externally with respect to proctype definitions are global variables that may be accessed by all the active processes. Channels for inter-process communications are defined like variables by means of the declaration

```
Chan <identifier> = [<size>] of {<type>}
```

Where the parameter *size* determines the capacity of the FIFO queue associated with the channel and *type* determines the types of data that may be transferred. A zero size means that the channel is a rendezvous port, i.e. a synchronous interaction port. Arrays of variables can be declared and used in PROMELA with pure C syntax. Interactions between two processes take place by having one process send a message and the other one receive it. These operations take the form of special statements:

- <channel-identifier> !<parameter-list>
- <channel-identifier>?<variable-list>

The first statement is the output operation that sends the data in the <parameter-list> (a list of comma separated expressions) to the channel specified by <channel-identifier>, whereas the second statement is the corresponding input operation that stores the received data in the specified variables. A special variable, denoted by an underscore, is used to indicate that the input value is to be discarded.

Statements in PROMELA may be executable or non-executable, according to the current status of the system. Predicates are statements as well; they are executable only if the predicate holds, but their execution has no effects on the system states. The main control statements are the *if* and *do* statement. The statement

```
if
:: <statement 1>
...
:: <statement n>
fi
```

represents an n-way selection in which one of the executable branches is selected randomly. The statement

```
do
:: <statement 1>
...
:: <statement n>
od
```

represents a loop in which at each iteration one of the executable branches is selected randomly. Loops terminate with *break* or *goto* statements.

In PROMELA, there is no difference between conditions and statements. Even isolated boolean conditions can be used as statements. The execution of a statement is conditional on its executability. Executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For example, a busy wait loop

While (a!=b) skip

Can be achieved in PROMELA with the statement

(a == b)

The condition can only be executed (passed) if it holds. If the condition does not hold, execution blocks until it does.

Processes may be instantiated dynamically by means of the statement

run <identifier> (<actual parameter list>)

Statements can be concatenated by the usual semicolon symbol or by the equivalent -> symbol.

The PROMELA language also lets one specify that a sequence of statements is to be executed atomically, by enclosing it in an *atomic* block with the syntax:

atomic {<sequence of statements>}

Within an atomic block only the first statement may be non-executable. Atomic blocks are exploited by the model checker to simplify the analysis. In fact, states corresponding to intermediate steps in an atomic block need not be represented.

Formalization of correctness criteria in PROMELA can be expressed in the following language constructs.

- The assert( ) statement, which can be used to express both local assertions and global system invariants
- Three types of labels that can be used to define a small class of frequently used correctness claims for terminating and cyclic sequences
- The formalization of general temporal claims
- The notation that can be used in assertions and in temporal claims to refer to the control-flow states and the local variables of arbitrary running processes

Chapter 6

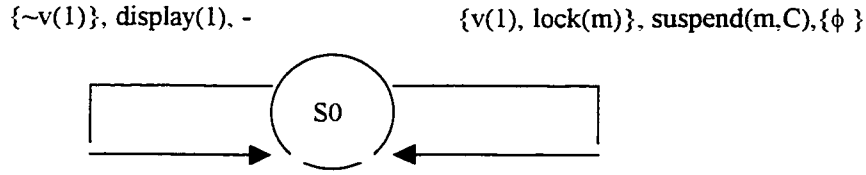
TRANSLATE NETWORK OF SFSMS INTO PROMELA

This chapter describes how to translate the design of network of synchronizing finite state machine into PROMELA through an example.

6.1 Problem Description

Suppose we have designed for such an application: there are two separate processes (proc1 and proc2), each doing some work and showing results to users. Users can suspend and resume each of these two processes through a graphical user's interface, which is kept active by the main process (mainProcess).

Figure 1 shows the SFSMs for the behavior of each of these three processes. Part (a) and (b) illustrates the behavior of *proc1* and *proc2* respectively. Each keeps checking the status of the related boolean variable (v(1) or v(2)). If it is true, it puts itself to the waiting queue of Java monitor m, otherwise, it does its own work and shows the result (denoted by action display(1) or display(2)). Part (c) in Figure 1 illustrates the behavior of the main process. It receives command from users to toggle between suspend and resume status for process proc1 (toggle(1)) or proc2 (toggle(2)), and toggle the value of the corresponding variable (v(1) or v(2)). This is followed by action *resume* to wake up a process on the waiting queue of Java monitor m.



(a)

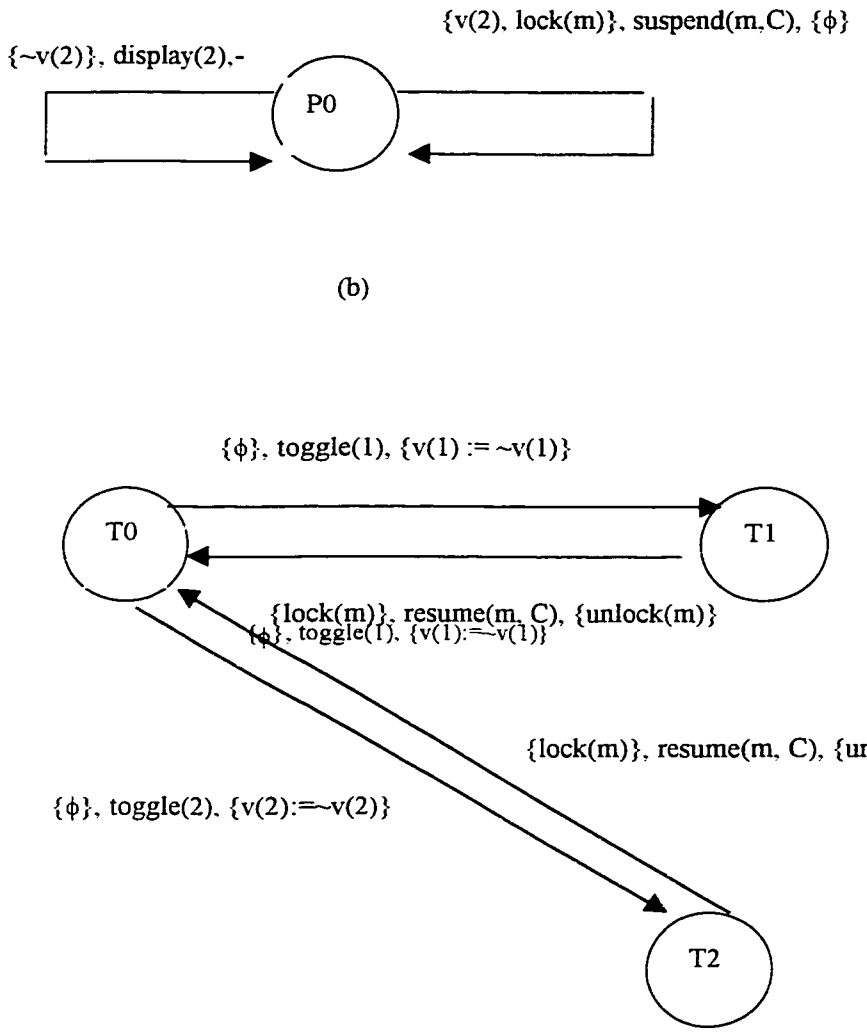


Fig. 6.1. SFSMs for each process in the suspend/resume example

6.2 Definition for RC_Pair in PROMELA

From the above description of transitions in our NSFSM model, we can see that RC_Pair plays an important role in the verification. The mutual exclusion and synchronization are fulfilled by the operations of RC_Pair. In our translation, the first thing is how to express RC_Pair mechanism in PROMELA.

The syntax of RC_Pair is based on encapsulating items of data and the procedures that operate upon them in a single module. The interface to a RC_Pair will consist of a set of procedures. These procedures operate on data that hidden within the module. A RC_Pair not only protects internal data from unrestricted access but also synchronizes calls to the interface procedures. The implementation ensures that the procedures are executed under mutual exclusion. For verification of concurrent(distributed) systems, we only care for synchronization among multiple processes and ignore the details of internal data. Following this idea, we can define a RC_Pair class in PROMELA which only contains the common attributes of all RC_Pairs. Whether the designer implements semaphore or any kind of monitors, we consider them in a identical form with different parameters. This definition is inspired by Klaus Havelund and Thomas Pressburger's translation of Java program into PROMELA [15]. Their work concentrate on translating a class *Buffer* with synchronized methods. Our approach is to give a general definition for all RC_Pairs in concurrent systems.

```
#define queueLength 5

typedef WAIT {
  int queue[queueLength];
};


typedef RCPair {
int numResource;
int numCondition;
int S;
WAIT waitingQueue[numCondition];
chan condWait[numCondition] = [0] of {byte};
int sp[numCondition];
int rp[numCondition];
};


#define MaxPair 10

RCPair pairs[MaxPair];

byte nextPair = 0;
```

Fig. 6.2 RC_Pair attributes

The general principle behind the translation is the following. A RC_Pair basically consists of data variables and procedures. For each new creation of a RC_Pair, a new set of data variables is allocated, and the procedures of that pair will then work on this newly allocated area. Hence, at any point in time a set of data areas will have been allocated, one for each pair. We model data areas of RC_Pairs by an array of records (typedef's in SPIN terminology), one record for each pair. An index variable (nextPair) will point to the next free record in the array, initially having the value 0 (first record in the array). Procedure Definitions are mapped into macro definitions, and procedure calls are mapped into applications of macros (Figure 6.2).

Now we will explain the PROMELA code. At first we define a constant *queueLength*, which indicates how many processes can be stored in the waiting queue of a condition in any pair. Users can define different values for this variable according to the particular system. Because our system does not contain many processes, we use 10 now. If we hope to locate a process in a waiting queue of any condition of a pair, we must know the index of the condition in the pair and the index of the process in the queue of the condition. Because SPIN does not support multi-dimensional array, we have to define a *WAIT* type first which contains an array *queue*. This type will be implemented in the type definition of *RCPair*. The first two attributes of RCPair indicates the pair is a semaphore or a monitor as we described in Chapter 2.

The attribute *S* is a variable that indicates the current number of resources that have not been locked. The initial value of *S* is equal to *numResource*. Every time a process locks a resource, the value of S will be decremented by 1. When a process releases a resource, the value of S will be incremented by 1. These operations are applicable to both busy-wait semaphore and monitors. We only need to note that in monitors, the initial value of S must be 1.

We know that in RCPair, every condition will have a waiting queue, so we define an array for the waitingqueues of all conditions. The next array lockPid will contain all the process Ids that have locked a resource. When a process locks a resource, its ID will be

put in the first slot whose value is 0 in the array lockPid. When the process releases the resource, the corresponding slot will contain 0 again.

Every process can decide to suspend itself on some condition and be resumed by other process later, so there must be communication mechanism between processes. For every condition, we define a rendezvous channel so that a process that occupies the RCPair can send *resume* message through the channel to the process that has suspended on this condition. All these channels are put in one array corresponding to the number of conditions.

The use of integer *Rcount* and integer array *sp, rp and Head* will be explained in the definition of macros later. We define a constant maxPair which tells the total number of RCPairs in a system. The user can assign a different number according to their need. All the RCPairs are stored in an array and can be accessed by their index in the array. The initial index value is 0.

```
#define RC_constructor(numR, numC) atomic {\
pairs[nextPair].numResource = numR; \
pairs[nextPair].numCondition = numC; \
pairs[nextPair].S = numR; \
nextPair ++ }
```

Figure 6.3 Constructor of RCPair

When we hope to add a new *RCPair* to the array, we will call the *RC_constructor*. Notice that we only provide two parameters and assign initial values to *numResource*, *numCondition* and *S*. The initial values of other attributes have default value 0. At the end, the nextPair will be incremented by 1 so that we can add next pair.

```
#define this _pid

#define RC_lock(RC_index) \
atomic {pairs[RC_index].S>0-> pairs[RC_index].S-- }

#define RC_unlock(RC_index) atomic { pairs[RC_index].S++ }
```

Fig. 6.4 Lock and unlock operations of RCPair

In PROMELA, _pid is a special constant that indicates the ID of current process. When a process hopes to lock a RCPair, it provides the index to locate the pair in the array, then checks if this pair still has resource to be locked (S>0). If the value of S is zero, the process will be blocked until S is changed. If S>0, the process will decrement S by 1. The unlock operation is opposite to the lock operation. When a process hopes to releases the RCPair it has locked, it provides the index and locates the pair. After this is finished, it will increment S by 1.

```
#define awake 0

#define RC_suspend(RC_index, cond_index) \
atomic { \
        RC_unlock(RC_index); \
        pairs[RC_index].waitingQueue[cond_index]. \
                queue[pairs[RC_index].sp[cond_index]]=this;  \
        pairs[RC_index].sp[cond_index] =(pairs[RC_index].sp[cond_index] +1) % queueLength; \
        pairs[RC_index].waitingQueue[cond_index].queue \
        [pairs[RC_index].rp[cond_index]] == this -> \
        pairs[RC_index].condWait[cond_index]?awake-> \
                        RC_lock(RC_index) \
        }


#define RC_resume(RC_index, cond_index) \
        atomic { \
        if \
        ::pairs[RC_index].waitingQueue[cond_index].queue[pairs[RC_index]. \
                rp[cond_index]] == 0 ->skip; \
        ::else->pairs[RC_index].condWait[cond_index]!awake; \
                pairs[RC_index].waitingQueue[cond_index].queue[pairs[RC_index] \
                .rp[cond_index]] = 0; \
                pairs[RC_index].rp[cond_index]=(pairs[RC_index].rp[cond_index] \
                + 1 ) % queueLength; \
        fi }

#define RC_resumeAll(RC_index, cond_index) \
        atomic { \
        do \
        ::pairs[RC_index].waitingQueue[cond_index].queue[pairs[RC_index]. \
                rp[cond_index]]==0 ->break; \
        ::else->pairs[RC_index].condWait[cond_index]!awake; \
                pairs[RC_index].waitingQueue[cond_index].queue[pairs[ \
                RC_index].rp[cond_index]]=0; \
                pairs[RC_index].rp[cond_index]=(pairs[RC_index].rp \
                        [cond_index] + 1) % queueLength; \
        od }
```

Fig. 6.5 Suspend and resume/resumeAll operations

We will illustrate the suspend and resume/resumeAll operations in Figure 6.6.

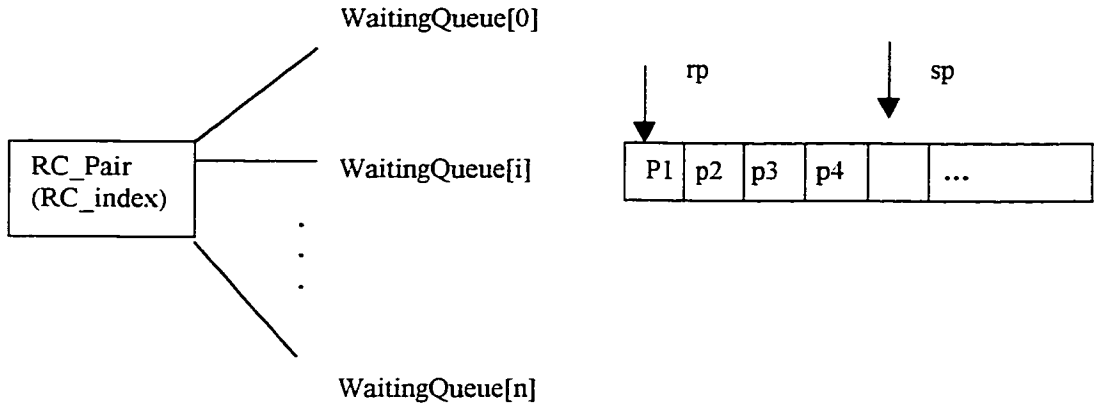


Fig. 6.6 Pointers sp and rp in waitingQueue of conditions

For suspend action, we need to provide two parameters: the index of RCPair and the index of the condition in this pair. When a process decides to suspend itself, it will release the lock and store its ID in the first zero slot (actually the position that the suspend pointer *sp* points to) of the waiting queue of the indicated condition. If the position is equal to the position that the resume pointer *rp* points to, that means the current process is the head of the waiting queue. *sp* will move forward one position in a round way after one process ID is stored in the waiting queue. When the suspended head process gets a message from the rendezvous channel, it will wake up and compete for the lock again. Only when a process becomes the head of the waiting queue, it can be resumed. For the *resumeAll* operation, we will let all processes in the waiting queue become head one by one in an atomic operation. If a process performs *resume* action, it will send a message to the head process in the waiting queue. If the head position of the waiting queue contains zero, we know that no process suspended on this condition and resume action does nothing. After the first process is resumed, its ID will be removed from the waiting queue and the resume pointer *rp* will move forward one position in a round way. *ResumeAll* action is similar to *resume*. The only difference is that resumeAll will resume all the processes in the waiting queue and we will change the values of *rp*

zero or more times. Note that we define 0 as the default value of every element in *queue*. Because the process ID of *init* process (in our translation, we will contain one *init* process in every PROMELA program) is always 0 and *init* process will not be involved in the operations of RC_Pair, there is no side effect in this definition.

6.3 Translation of Processes

We describe the behavior of each process in terms of Synchronizing Finite State Machine. On the other hand, the process type (proctype) in PROMELA models the transition from one state to another. Intuitively, we will translate each process into one process type. The proctype begins from initial state of the process and follows all possible transitions. For the start of every state, we assign a label to it (the label for the initial state is state0). Transition (s, (g, (a), ass), s') means that (I) the process currently in state s is capable of doing action *a* ending in state s', if guard g is satisfied by the environment (maybe include some lock of RCPair); (II) action *a* must be followed by the assignments expressed by post assignments (including unlock some RCPair) *ass*. (III) after this action is performed, the execution will go to the state *s'*. According to this interpretation of transition, our translation for any transition from state s to state s' will be like this (1) first we translate *g* into PROMELA statements. Note that in PROMELA, there is no difference between conditions and statements. If g is satisfied, the corresponding statements will be executable, otherwise they will be blocked; if g contains lock operation, find the particular pair in array pairs[]. (2) find the RCPairs occupied by action *a* in RCPair array pairs[]. (3) If action *a* calls some macros of the pair, pass the required indices as parameters, otherwise, just print the action *a* into corresponding statements; (4) Translate *ass* into assignment statements in PROMELA and go to the label for *s'*. All assignment and goto statements are executable. If *ass* includes *unlock() operation*, just call *RC_unlock* macro. Starting from one state, we may have several transition. For example, in process proc1, from state R0, we have two different transitions ending in R0. We use *if* structure to simulate these transitions. Each selection of *if* structure represents one transition. Which selection will be executed depends on which guard is satisfied. For each state, we have one *if* structure.

Communication among processes in our design is fulfilled by the change of global variables (the environment). When one process changes the value of some global variables, the related process will take corresponding actions. In this example, there are three global variables: v1, v2 and toggle. Figure 6.6 shows the translation of proc1 and proc2.

```
bit v1=0;
bit v2=0;
byte toggle=0;

proctype proc1 ()
{
S0: if
        :: v1 == 1->
                RC_lock(0);
                RC_suspend(0, 0);
                goto S0;
        :: v1 == 0 ->
                printf("Display(1)");
                goto S0;
        fi
}
proctype proc2 ()
{
P0: if
        :: v2 == 1 ->
                RC_lock(0);
                RC_suspend(0, 0);
                goto P0;
        :: v2 == 0 ->
                printf("Display(2)");
                goto P0;
    fi
}

proctype mainProcess()
{
T0: if
        :: toggle = 1;
                printf("toggle(1)");
                v1= ~v1;
                goto T1;
        :: toggle = 2;
                printf("toggle(2)");
                v2 = ~v2;
                goto T2;
   fi
T1:   if
        :: RC_lock(0);
```

```
        RC_resume(0, 0);
        RC_unlock(0);
        goto T0;
        fi

T2:     if
        :: RC_lock(0);
          RC_resume(0,0);
          RC_unlock(0);
          goto T0;
        fi
}
```

Fig. 6.7. Process proc1, proc2 and mainProcess

In every system, we must have an *init* process to construct the RC_Pairs and run all the processes. In order to start all the processes at the same time, we put all the run statement in an atomic sequence (Figure 6.7). Because we hope to start the execution of all the processes at the same time, we put all the *run* statements in an atomic sequence. We will explain the monitor process later.

```
init {
        RC_constructor(1,1);
        atomic {run mainProcess(); run proc1(); run proc2(); run monitor(); }
}
```

Fig. 6.8 Init process

6.4 Correctness Requirement

6.4.1 Assertion

The correctness of concurrent system is imperative to formalize. In PROMELA, the assertions are often used to formalize system invariants such as boolean conditions that, if true in the initial state, remain true in all reachable states, independent of the execution sequence. During translation, we must analyze the correctness properties that we are going to check and express them in boolean conditions. These assertions can be translated into PROMELA statement *assert(condition)* that are always executable and can be placed anywhere in the translated process. If the assertion is violated during the execution, at least one execution sequence will result in concurrency-related errors.

In our example, we know that the RC_Pair involved in the system is a monitor with only one condition. We have an invariant that at one time, at most one process can occupy the monitor. The value of S will be always between 0 and 1. We put all the assertion statement in a new process called "monitor" (Figure 6.8).

```
proctype monitor() {assert(pairs[0].S>=0 && pairs[0].S<=1) }
```

Fig. 6.9 Monitor process for assertion statements

6.4.2 Deadlocks

In order to define what a deadlock in a PROMELA model is, we must distinguish the expected, or proper, end-states from the expected ones. The unexpected end-states will include not just deadlock states, but also many error states that are the result of a logical incompleteness of the protocol specification.

The final state in a terminating execution sequence must minimally satisfy the following two criteria to be considered a proper end-state:

- Every process that was instantiated has either terminated or has reached a state marked as a proper end-state
- All message channels are empty

In our current example, we do not consider the deadlock problem. In later chapter, we will give an example about deadlock. Any final state in a terminating execution sequence that does not satisfy the two criteria for proper end-states is automatically classified as an improper end-state.

6.4.3 Bad Cycles

Two properties of cyclic sequences can be expressed in PROMELA, corresponding to two standard types of correctness requirements. Both properties are based on the explicit marking of states in a validation model. The first property specifies that

- There is no infinite cycle through unmarked states.

The marked states are called progress-states, and the execution sequences that violate the above correctness claim are called non-progress cycles. A progress-state label marks a state that must be executed for the protocol to make progress. In our example, we define some progress states.

The second property is the opposite of the first. It is used to specify that

- There is no infinite behaviors that include marked states

Execution sequences that violate this claim are called livelocks. We define acceptance-state labels. An acceptance-state label is any label starting with the character sequence "accept." It marks a state that may not be part of a sequence of states that can be repeated infinitely often.

6.4.4 Temporal Claims

Temporal claims define temporal orderings of properties of states. To express the requirement that "every state in which property P is true is followed by a state in which property Q is true," we could write

P->Q

The requirement above can be expressed in PROMELA as

```
never {
        do
        :: skip
        :: break
        od ->
        P->!Q
}
```

That is, independent of the initial sequence of events, it is impossible for a state in which property P is true to be followed by a state in which property Q is false. The claim is matched, and the corresponding correctness property thereby violated, if and when the claim body terminates. In our example, we use temporal claim to define that when mainProcess toggles a process, the toggled process should be executed.

```
        never {
        do
        :: skip
        :: toggle == 1 -> goto accept1
        :: toggle == 2 -> goto accept2
        od;
accept1:
        do
        :: !proc1[2]:S0
        od;
accept2:
        do
        :: !proc2[3]:P0
        od;
}
```

Fig. 6.10 Temporal claim

So far we have introduced all the rules to automatically translate network of synchronizing finite state machines into PROMELA programs. In later chapters, we will apply these rules to solve some other problems.

Chapter 7

IMPLEMENTATION FOR THE AUTOMATIC TRANSLATOR

In this chapter, we present our implementation for the automatic translator. Based on the translation rules we presented in the previous chapter, we developed a software called "Translator". Object-oriented design and programming methodologies are utilized during the development. ArgoUml is used to help the software design and Java is chosen as the coding language.

7.1 Design of "Translator"

The two main tasks of "Translator" are to provide a friendly interface for users to input their design and to translate the input design into the PROMELA program. Let us take a look at the interface first. Figure 7.1 shows the graphical user interface.

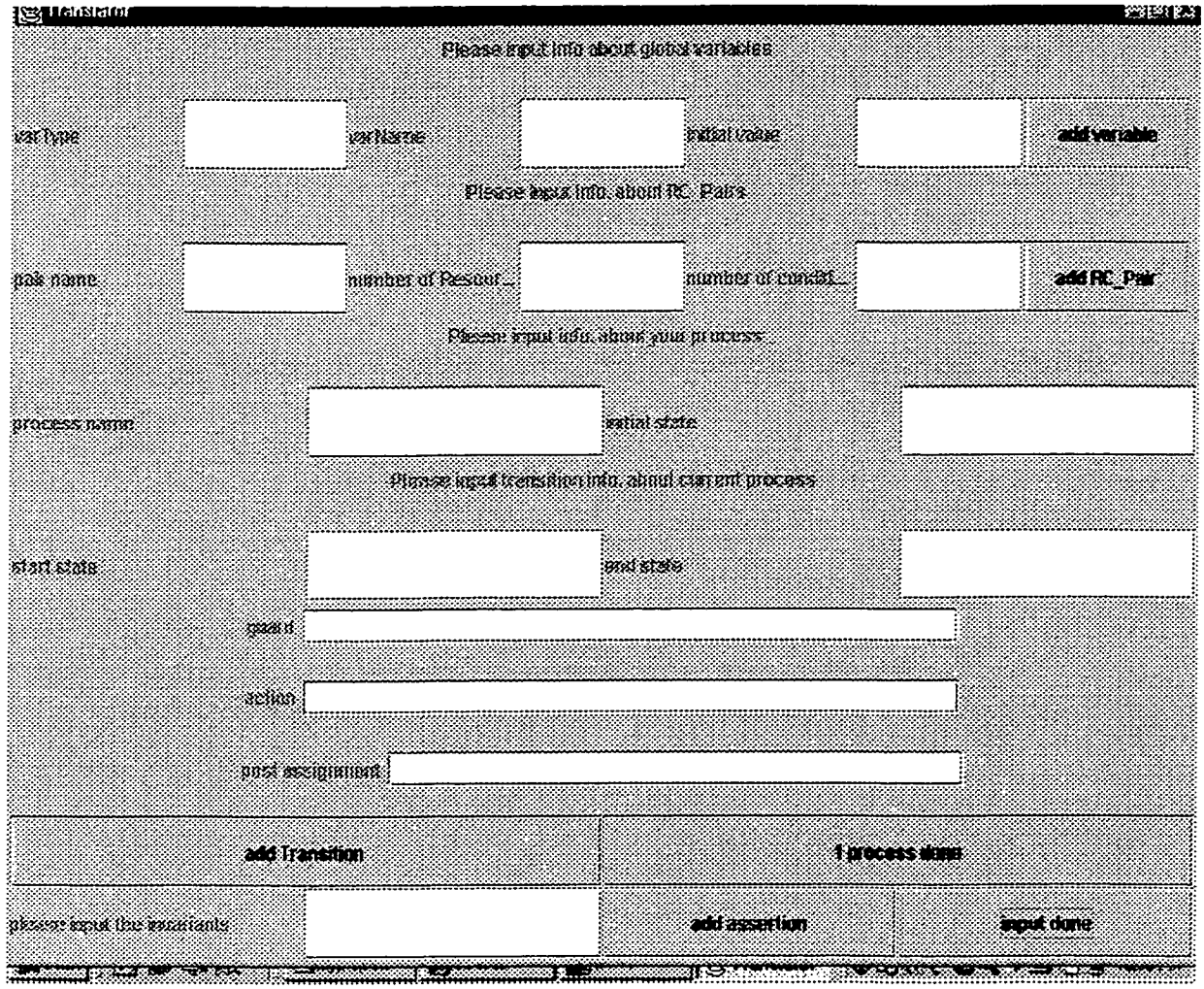


Fig. 7.1 The interface of "Translator"

From Figure 4.2, we can see that the network of synchronizing finite state machine (NSFSM) is composed of three parts: the global variables, the RCPairs and the synchronizing finite state machines. The correctness requirements can be expressed as assertions. The translation work actually is to express these four parts in suitable PROMELA statements. Our interface can be divided into four areas corresponding to the four parts. Now we will introduce them step by step.

The information about global variables is entered in the top area. The user can input the type, name and initial value of one global variable in the text field, then click the "add variable" button. The action connected to the button will be performed and the variable information is stored by the translator. The user can input all the global variables one by one in this area and the information is stored in a vector.

The second area is the information about RCPairs. It works in a similar way to the global variable area. Note that for each RCPair, we need to give three parameters: the name of the RCPair, the numResource and the numCondition. All the RCPairs are stored in a vector, too.

The third area is the most complicated area. Users can enter information about all their processes. Each process contains a name, an initial state and some transitions. For a given process, we need to input all the transitions in this process and then click the button "1 process done". For each transition, we will input the starting state, the end state, the guard, the action and the post assignments. Note that we do not let users input the information about states because this kind of information can be extracted from transitions, every transition contains one starting state and one end state. Later we will explain how to do it in the algorithm.

The last area is the assertion part, which is very easy to understand. Users can input their invariants in the text field one by one and the translator will put them in a particular process "monitor()".

The next step is to define the use cases. Figure 7.2 shows the use case diagram for our translator.

The actor is the user, who will use "translate NSFSM". The use case "translate NSFSM" depends on four use cases "translate global variable", "translate RCPairs", "translate process" and "translate assertion". "translate process" depends on "translate state" and "translate state" depends on "translate transition".

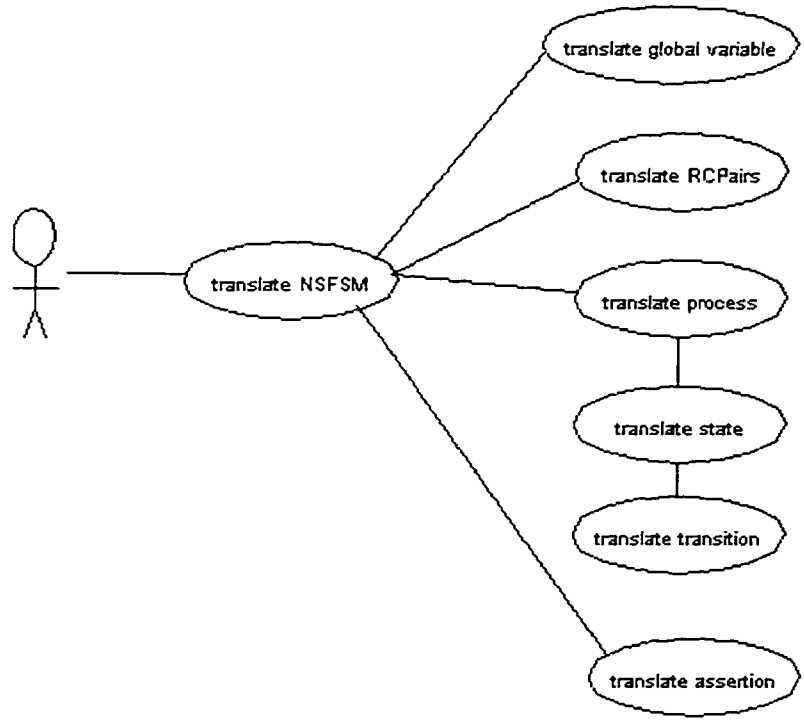


Fig. 7.2 Use case diagram

Based on the analysis of graphical user interface and the use case diagram, we can draw the class diagram (Figure 7.3).

There are six classes in the translator: *State*, *Transition*, *RCPair*, *SFSMprocess*, *GlobalVariable* and *Translator*. *Translator* contains *main* method and perform the *translate* operation. *Translator* is composed of *SFSMprocess*, *RCPair* and *GlobalVariable*. Assertions are strings, so we do not need to define *assertion* classes. Note that *SFSMprocess* contains *State* and *Transition*. Each state contains a vector of transitions that start from the current state.

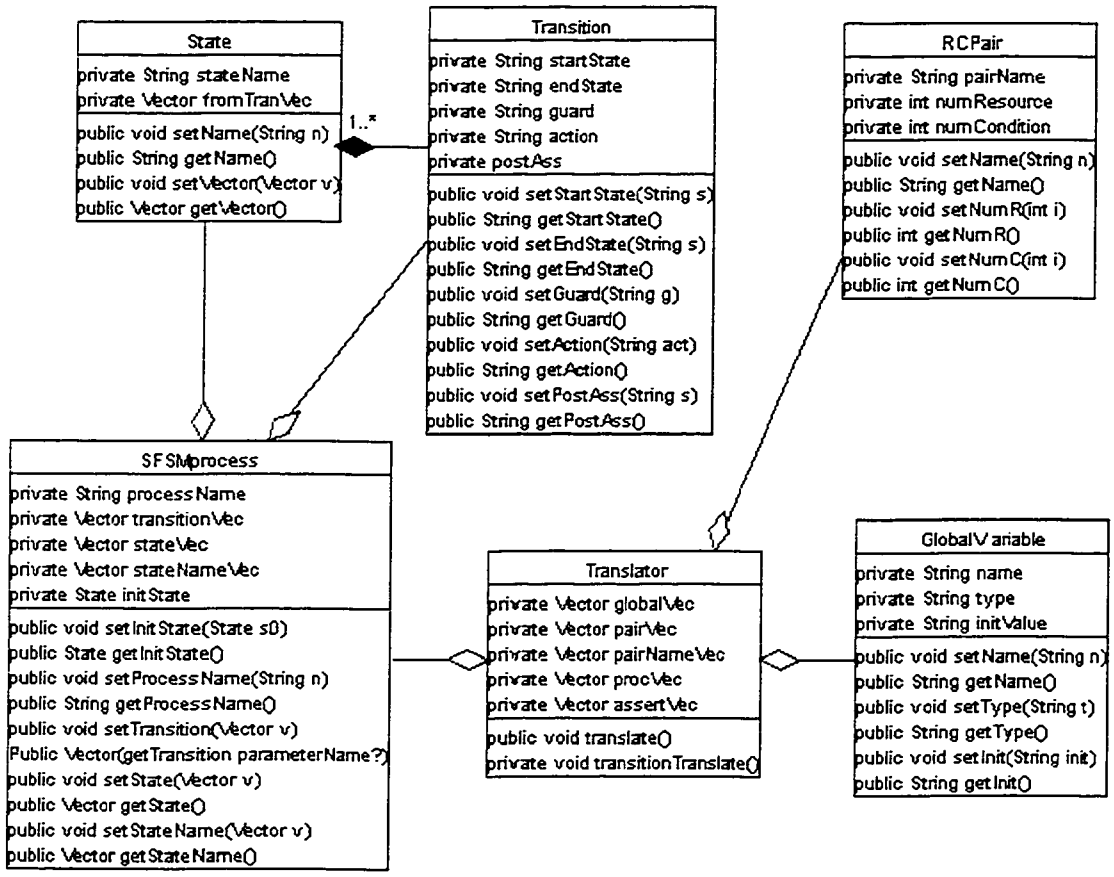


Fig. 7.3 Class diagram

7.2 Algorithm

The algorithm for the translator is presented in Figure 7.4. There are three procedures in the algorithm: getInput, translate and transitionTranslate.

Input: a list of global variables;
A list of RCPairs;
A list of SFSMprocesses (Each process contains a list of transitions);
A list of assertions;

Output: a PROMELA program.

Procedure getInput()

1. Store all global variables in GlobalVec.
2. Store all RCPairs in PairVec and all pair names in pairNameVec.
3. For each process, store all transitions in TransitionVec.
   3.1 Scan TransitionVec to get the names of all states in this process and store the state names in stateNameVec.
   3.2 For every state name in stateNameVec, scan the TransitionVec if the state name is equal to the starting state of a transition, add this transition to the fromTranVec of the state.
   3.3 Store all the states in the StateVector of the process.
4. Store all the processes in procVec.
5. Store all the assertions in assertVec.

Procedure translate()

1. print the information about global variables into StringBuffer.
2. While (procVec has more elements)

```
{
    process := next element;
  print process name to StringBuffer;
  while (stateVec of process has more elements)
  {
            state := next element;
            print if structure to StringBuffer;
            while (fromTranVec of state has more elements)
            {
                    call transitionTranslate();
            }
   }
   }
```

3. print monitor() process to StringBuffer;
   print assertion Strings to StringBuffer;
   print *init* process to StringBuffer;
   print RCPair constructor to StringBuffer;
   print *run* statements to StringBuffer;

procedure transitionTranslate()

1. decompose guard into separate preconditions.
2. If (a precondition contains "lock(")
   {
   locate the index of RCPair in pairVec and perform the RC_lock operation
   }
   else {print guard to StringBuffer }
3. if (action contains "suspend(" or "resume(" or "resumeAll(")
   {
       locate the indices of RCPair and condition, perform the corresponding operations
   }
   else {print action to StringBuffer }
4. decompose post assignments into separate statements
5. if (possAss contains "unlock(")
   {
       locate the index of RCPair in pairVec and perform the RC_unlock operation
   }

else {print postAss to StringBuffer }

Fig. 7.4 Algorithm for translator

Based on the above algorithm, we can develop Java code for automatic translation. The source code is presented in the Appendix.

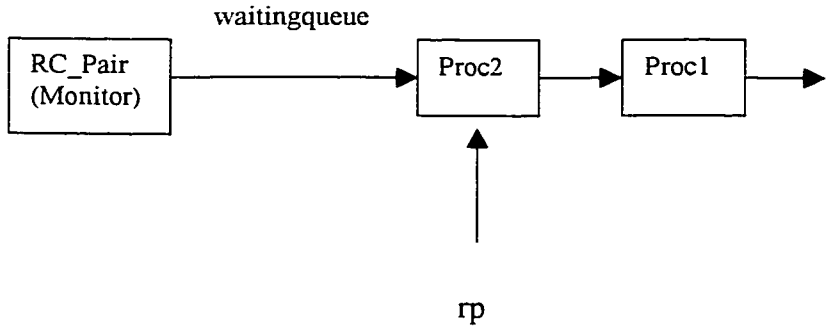
Chapter 8

EVALUATION FOR MODEL CHECKING WITH SPIN

In this chapter, we present an analysis and experimental results to evaluate our design notation and translation rules from network of SFSMs. In section 1, we analyze the suspend/resume example we presented in previous chapter. In section 2, we describe two other examples and show the translation results. In section 3, we give the summary.

8.1 Analysis of Suspend/Resume Problem

The network of SFSM design and translation into PROMELA of the suspend/resume problem has been presented in Chapter 6. We know that the mainProcess will toggle between process1 and process2. After both processes are in the waiting queue of the monitor m, the mainProcess tries to wake up one of them by calling resume operations. Because resume operation only wakes up the process in the head position at the waiting queue, the process being toggled may not be in the head position and continue to be in waiting state.



The mainProcess toggles process1, but process2 is resumed.

Fig. 8.1 An erroneous state in suspend/resume problem

After we run the PROMELA program, we find that the temporal claim (Fig. 6.10) is violated which means we reached an erroneous state similar to the one in Figure 8.1.

8.2 Two Other Applications

8.2.1 Producer-Consumer with bounded buffer problem

Now let us see the producer-consumer problem we mentioned in Chapter 2.

There are 4 global variables:

int in = 0	//the position of next space, initial value is 0
int out = 0	//the position of next item, initial value is 0
int size = 5	//the size of the buffer
int count = 0	//the number of items in the buffer

There is one monitor (m) with two conditions

Condition 0: not_Full

Condition 1: not_Empty

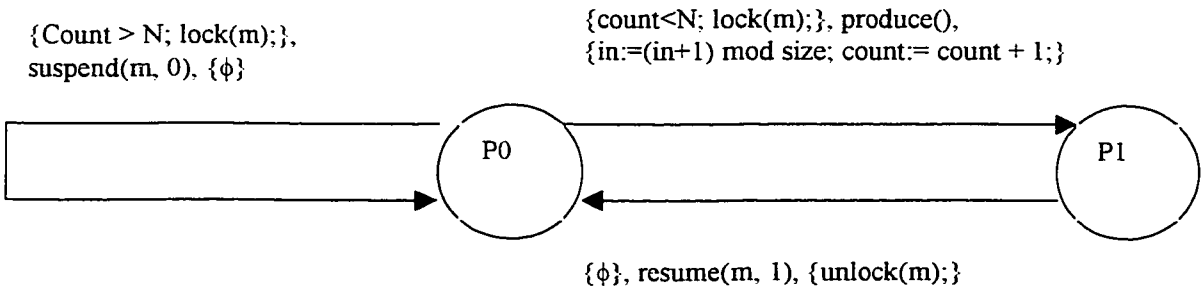


Fig. 8.2. Producer Process

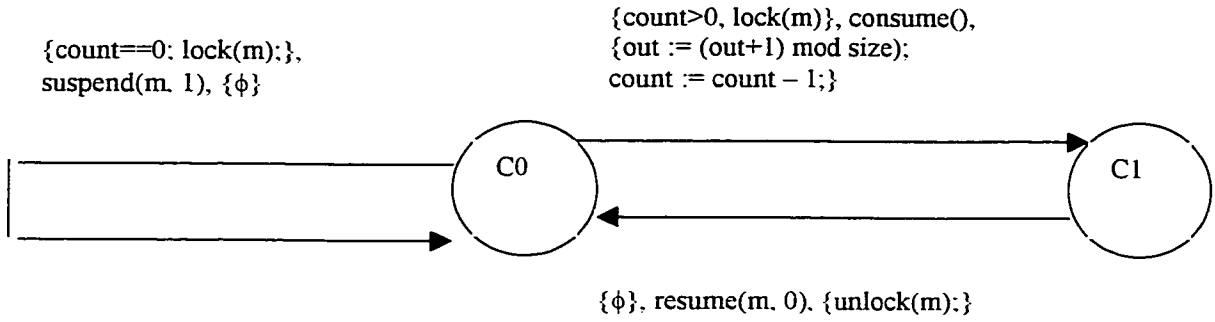


Fig. 8.3. Consumer Process

According to the rules we introduced in Chapter 6, the definition for RC_Pair and its operations are the same for every problem. We only need to translate the behavior of the processes into PROMELA.

```
int in =0;
int out = 0;
int count = 0;

#define size 5;

proctype producer()
{
state0: if
          :: count > N ->     /* this statement should be count == N-> */
            RC_lock(0);
            RC_suspend(0, 0);
            goto state0;
          :: count < N ->
            RC_lock(0);
            printf("produce()");
            in = (in + 1) % size;
            count = count + 1;
            goto state1;
state1: if
            :: RC_resume(0, 1);
              RC_unlock(0);
              goto state0;
            fi
          fi
}

proctype consumer()
{
```

```
state0: if
        :: count == 0 ->
           RC_lock(0);
           RC_suspend(0, 1);
           goto state0;
        :: count > 0 ->
           RC_lock(0);
           printf("consume()");
           out = (out + 1) % size;
           count = count - 1;
           goto state1;
state1: if
        :: RC_resume(0, 1);
           RC_unlock(0);
           goto state0;
        fi
    fi
}

proctype monitor() {assert(count >= 0 && count <= 5);}

init {
    RC_constructor(1, 2);
    atomic { run producer(); run consumer(); run monitor();} }
```

Fig. 8.4. Translation of Producer-consumer

When we check this program with SPIN, we find that the assertion is violated because the value of count can be 6. By tracking the execution, we find there exists an error in the guard of producer, as indicated in the program.

8.2.2 Dining Philosophers Problem

The Dining Philosophers problem is a well-known example, often used as a case study for concurrent programming. Our dining philosophers example involves five philosophers sitting in a circle. Each of the philosophers alternates between eating and thinking. To eat, philosophers must acquire two forks, each of which they share with one of their neighbors. All philosophers must pick the fork from their right first, then the fork from their left. All philosophers are implemented as processes and the forks are represented by RC_Pairs (semaphores actually).

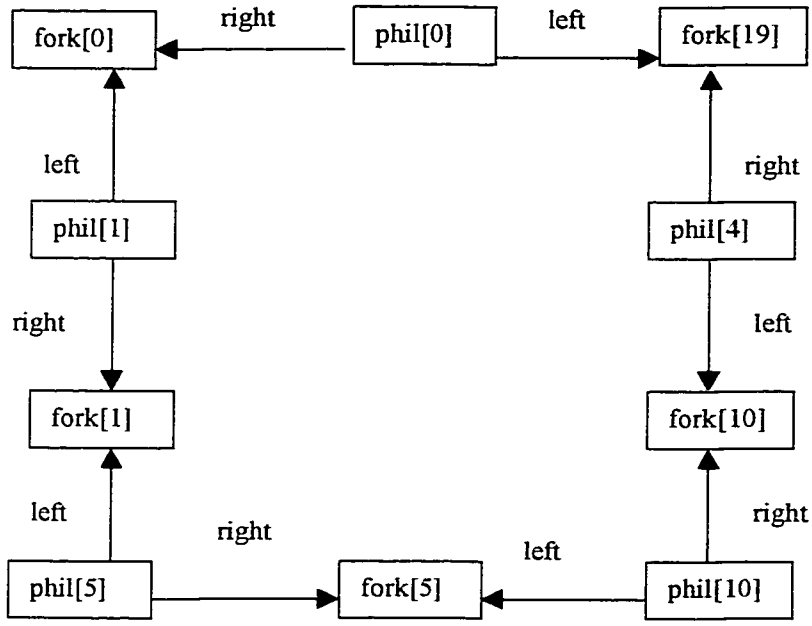


Fig. 8.5 Dining Philosophers composite model

5 semaphores: fork [0-4]

5 processes: phil[0-4]

5 global variables: hungry[0-4]

{hungry[j] == false;}, think(),
{hungry[j] = true;}

{hungry[j]==true; lock(fork[j]); lock(fork[(j+4) mod 5]);},
eat(), {(unlock(fork[(j+4) mod 5];unlock(fork[j]);
hungry[j] = false;}

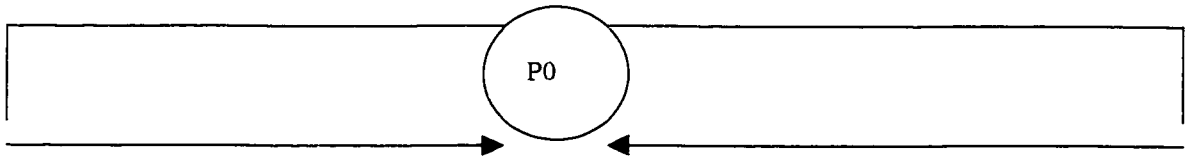


Fig. 8.6. Process phil[j], j=0...4

It is well known that if the five philosophers are hungry at the same time and pick up the fork at their right side at the same time, there will be a deadlock. Now we will translate

the five processes into PROMELA and run the program. The result indicates that there exists a deadlock.

```
bool hungry[5] = true;

proctype phil0 () {
state0:
if
::hungry[0]->RC_lock(4);
              RC_lock(0);
              printf("phil 0 is eating\n");
              RC_unlock(0);
              RC_unlock(4);
              hungry [0] = false;
                    goto state0;
::!hungry[0]->printf("phil 0 is thinking");
              hungry[0] = true;
                    goto state0;
fi}

proctype phil1 () {
state0:
if
::hungry[1] -> RC_lock(0);
               RC_lock(1);
               printf("phil 1 is eating\n");
               RC_unlock(1);
               RC_unlock(0);
               hungry[1] = false;
                     goto state0;
::!hungry[1] -> printf("phil 1 is thinking");
               hungry[1] = true;
                     goto state0;
fi
}

proctype phil2 () {
state0:
if
::hungry[2] -> RC_lock(1);
               RC_lock(2);
               printf("phil 2 is eating\n");
               RC_unlock(2);
               RC_unlock(1);
               hungry[2] = false;
                     goto state0;
::!hungry[2] -> printf("phil 2 is thinking\n");
                    hungry[2] = true;
                    goto state0;
fi}

proctype phil3 () {
state0:
```

```
if
::hungry[3] -> RC_lock(2);
               RC_lock(3);
               printf("phil 3 is eating");
               RC_unlock(3);
               RC_unlock(2);
               hungry[3] = false;
                    goto state0;
::!hungry[3]-> printf("phil 3 is thinking\n");
                    hungry[3] = true;
                     goto state0;
fi}

proctype phil4 () {
state0:
if
::hungry[4] -> RC_lock(3);
               RC_lock(4);
               printf("phil 4 is eating\n");
               RC_unlock(4);
               RC_unlock(3);
               hungry[4] = false;
                    goto state0;
::!hungry[4] -> printf("phil 4 is thinking\n");
                    hungry[4] = true;
                    goto state0;
fi}

init {
        RC_constructor(1, 0);
        RC_constructor(1, 0);
        RC_constructor(1, 0);
        RC_constructor(1, 0);
        RC_constructor(1, 0);

        atomic {
                run phil0(); run phil1(); run phil2(); run phil3(); run phil4() };}
```

Fig. 8.7 Translation of philosophers processes

8.3 Summary

The main purpose of our approach is to provide some new design notation (Network of Synchronizing Finite State Machine) that is easy to understand and master for designers and programmers, and a set of translation rules so that automatic translation from NSFSM to PROMELA can be performed. By checking the correctness of the three concurrent systems, we can see that our design notation and translation rules provide an effective way to do model checking with SPIN.

Chapter 9
Related Work

This chapter gives a brief review of earlier work on reproducible testing and static analysis of concurrent systems, including related work on reproducible testing control for nondeterminism, formal verification of concurrency, flow analysis for verification and model checking of concurrent Java Programs.

9.1 Reproducible Testing of Concurrent Programs

One of the biggest problems in regression testing of a concurrent program is nondeterminism which may produce different results from previous test with same test data. To solve this, some researchers have proposed some replay methods for reproducible testing. This section reviews two of them.

9.1.1 Synchronized-Communication Primitives

H. Sohn, D. Kung, etc. [25] proposed a set of synchronized-communication primitives and a new replay algorithm which uses these primitives to arrange all the execution sequences of non-deterministic operations. The communication primitives are the implementations of 1) Send_Request 2) Receive-Request 3) Send_Permit 4) Receive_Permit 5) Send_Complete and 6) Receive_Complete. The replay algorithm uses the communication primitives to communicate and control operations and shared variable access operations. This method significantly reduces the number of control processes and communication channels.

9.1.2 Language-Based Replay Control Method

Carver and Tai [5] proposed a language-based deterministic execution method using concurrent constructs such as queued semaphores and monitors for replay of concurrent programs. Using the synchronization and mutual exclusion constructs of the semaphores

and monitors, they can control the execution sequence of synchronization events by transforming the concurrent program into another program in the same language. It performs in two steps: 1) Collecting the sequences of synchronization events of a concurrent program by transforming it into a new program and executing the new one. 2) Controlling the execution of a concurrent program by transforming it into different programs that can replay the collected synchronization sequences.

9.2 Formal Verification Techniques

As we indicated before, concurrent systems are usually much more complex mainly due to the concurrency control and thread synchronization. In contrast to testing, software verification under certain formalisms and methodologies usually gives us higher confidence.

Much research work have been done to verify the correctness of concurrency related properties by formally specifying the requirements in a formula of a modal/temporal logic and describing system behavior in process algebra ([7], [10], [23]). However, formal verification techniques are hard to apply to most software systems mainly because the underlying specification languages are usually very difficult for ordinary designers and developers to understand and use. To overcome this difficulty, some semiformal/informal and especially graphical design notations are introduced into formal descriptions by some researchers. The most typical work is studying the translations of Unified Modeling language (UML) [3] into various formalisms. UML is well accepted by the software industries as graphical design notations. Here we give two examples: in [12], the formalism of class diagrams in UML using Z [26] has been discussed; in [21], operational semantics is provided to UML State Machine so that formal verification techniques can further apply. However, some parts of software systems such as the coordination among multiple processes, can hardly be clearly expressed by UML. In contrast to UML, SDL [30] provides us with both formal and graphical design notations, where process synchronization can be clearly expressed.

9.3 Data Flow Analysis Methods

This section reviews two types of data flow analysis methods which have the potential to provide cost-effective analysis of concurrent programs with respect to explicitly stated correctness properties.

9.3.1 Data Flow Analysis for Programs with rendezvous communication

Matthew B. Dwyer and Lori A. Clarke [9] describe an approach based on data flow analysis. Using this approach, a developer specifies a property of a concurrent program as a pattern of selected program events and asks the analysis to verify that all or no program executions satisfy the given property. They developed a family of polynomial-time, conservative data flow analysis algorithms that support reasoning about these questions. A prototype toolset is implemented that automates the analysis for programs with explicit tasking and rendezvous style communication.

9.3.2 FLAVERS Data Flow Analysis Technique for Concurrent Java Programs

G. Naumovich, G. S. Avrunin and L. A. Clarke [24] describes how the FLAVERS (Flow Analysis for VERification of Systems) data flow analysis technique, originally formulated for programs with the rendezvous model of concurrency, can be applied to concurrent Java programs. The general approach of FLAVERS is based on modeling a concurrent program as a flow graph and using a data flow analysis algorithm over this graph to check statically if a property holds on all executions of the program. This paper presents a straightforward approach for modeling Java programs that uses the accuracy improving mechanism to represent the possible communications among threads in Java programs, instead of representing them directly in the flow graph model.

9.4 Model Checking of Concurrent Java Programs

This section reviews some research work on static analysis approaches of concurrent Java programs. Model checking is the main static analysis method. The following two approaches both construct formal models and use the model checker SPIN to do model checking.

9.4.1 Static Analysis of Java Concurrency Mechanism

The Java language enables the development of concurrent and distributed software through the concepts of thread and remote method invocation. C. Demartini and R. Sisto [8] provide formalizations of the Java concurrency mechanism that enable the use of static analysis techniques similar to the ones used for the Ada language. This paper extends the approach followed for Ada tasking programs to the new Java language, providing formal models for the main thread synchronization primitives offered by the language. The formalism used is Promela and the input language of the model checker is SPIN.

9.4.2 Applying the Java PathFinder

Verifying programs is different from verifying hardware or protocols: the state space is often much bigger and the relationships harder to understand. Klaus Havelund, etc. [14] [15] present their experiences in applying the Java PathFinder (JPF), a Java to Promela translator abstraction of a program by hand and translated the simplified Java program to PROMELA using JPF. JPF allows a programmer to annotate the Java program with assertions and verify them using the SPIN model checker. In addition, deadlocks can be identified.

9.5 Advantage of Our Approach

Due to concurrency control and thread synchronization, certain properties that usually should hold for concurrent systems, such as invariance, eventualities, fairness etc. are by nature hard to test. In contrast to testing, static analysis under certain formalisms and methodologies usually gives us higher confidence in our conclusion. In our approach, we model the concurrent system design as a network of synchronizing finite state machines and use static analysis to ensure the correctness.

Compared with formal verification techniques that require difficult specification language, our innovation lies in introducing semaphore and monitor synchronization mechanisms directly into the design stage. One of the advantages in doing so is that in some cases we are able to express the communications among processes when the partner processes in the communications cannot be statically determined. Of course, in such a situation, we can also follow some other approaches like SDL, by explicitly modeling the synchronization mechanisms like monitors. However, this adds additional burdens for designers while the modeling of such synchronization mechanism does not need to be implemented at all. So it is better that we implicitly embed such mechanism into the underlying formal model and thus take it into account during the automated verification.

Data flow analysis has been applied to verify limited properties of concurrent programs. These methods are useful for ruling out certain behaviors of the concurrent system but are less precise than model checking. Our approach uses the existing model checking tool SPIN to verify whether the given model satisfies the considered properties.

Unlike applying JPF to model check concurrent Java programs, we do the model checking at the design stage. It is well known that finding concurrency errors in design stage and modifying them will explicitly reduce the cost of software development. On the other hand, for model checking with SPIN which is based on finite state machine, the state space of the final code is much bigger than the design specification, thus making it more difficult to verify.

Chapter 10
CONCLUSIONS AND FUTURE WORK

This chapter draws the conclusions and describes one possible direction for future work related to this research.

10.1 Conclusions

In this thesis, concurrent (distributed) systems and non-deterministic behavior are first introduced. Then we reviewed two mutual exclusion and synchronization mechanism: semaphores and monitors. We analyzed and compared the two mechanisms. Based on these analyses, we applied the design notation - Network of Synchronizing Finite State Machine (NSFSM). NSFSM can be applied in the design of concurrent (distributed) systems, which contain a new concept - RC_Pair. RC_Pair is a combination of semaphores and monitors, which contains both mutual exclusion and synchronization mechanisms. Through RC_Pairs, we include concurrency-related controls in NSFSM design. RC_Pair is a concept at the abstract level, which means we do not need to consider the particular implementation language. Programmers can choose any language they like to implement the design.

For the new design notation we proposed, we provide a set of automatic translation rules from NSFSM to PROMELA in order that the users can easily perform formal verification of NSFSM with SPIN. Following these rules, the users can translate their NSFSM design into PROMELA programs without much difficulty even though they are not familiar with SPIN and PROMELA. SPIN is a powerful and popular model checker. After we get the PROMELA program, SPIN can help us analyze whether there exist concurrency-related errors in our design.

The main contributions of this thesis are:

(1) Presenting concurrent system architecture and non-deterministic behavior involved in concurrent systems.

(2) Analyzing and comparing the two mutual exclusion and synchronization mechanisms: semaphores and monitors.

(3) Proposing a new concept RC_Pair which combines both semaphore and monitor mechanisms.

(4) Providing a set of automatic translation rules from NSFSM to PROMELA.

(5) Providing the design of NSFSM and implementation of automatic translation rules.

(6) Showing the results and evaluating our approach.

10.2 Future Work

In this thesis, we proposed our research about the new concept RC_Pair in NSFSM and automatic translation rules from NSFSM to PROMELA. According to the process of formal verification, some extensions to this research remain for further exploration.

To learn and master new concepts and tools for ensuring the correctness of concurrent systems is always a heavy burden for ordinary designers and programmers. For example, if the users hope to do formal verification with SPIN, they must master some design notation and know how to express their design in PROMELA. In addition, they must have a SPIN compiler in their machines and know how to run SPIN. A promising way to reduce this burden is to provide a thin client environment through the Internet. By using web technologies, such as Java Server Page, Enterprise Java Bean, etc, we can provide a friendly interface to let users input their design. The translation of designs into PROMELA and the model checking work with SPIN are all performed at the server side. At last, the server can send the model checking results to the client. In this way, the client does not need to know much about SPIN and PROMELA.

REFERENCE:

[1] G. Avrunin, J.C. Corbett and M. B. Dwyer; "Comparing Finite-State Verification Techniques for Concurrent Software"; Department of Computer Science, University of Massachusetts, MA, Nov. 1999

[2] M. Ben-Ari, "Principles of Concurrent and Distributed Programming", Prentice Hall, 1990.

[3] G. Booch, I. Jacobson, and J. Rumbaugh; "The Unified Modeling Language User Guide"; Addison-Wesley, 1998.

[4] G. Boudol, R. de Simone, V. Roy, and D. Vergamini; "Process calculi, from theory to practice: Verification tools"; In Proc. Of Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407. Springer-Verlag, 1990.

[5] R. H. Caver and K. C. Tai; "Replay and Testing for Concurrent Programs"; IEEE Software, 8(3):p66-74; Mar. 1991.

[6] Jessica Chen; "A Study on Static Analysis in Network of Synchronizing FSMs"; Proc. of the 7th Asia-Pacific Software Engineering Conference ({APSEC} 2000), page 489-493, IEEE Computer Society Press, 2000.

[7] R. Cleavaland and S. Sims. "The NCSU concurrency workbench"; In Computer-Aided Verification (CAV'96), LNCS 1102, pages 394-397. Springer-Verlag, 1996.

[8] Claudio Demartini and Riccardo Sisto, "Static analysis of Java multithreaded and distributed applications", Proceedings of the 1998 International Workshop on Software Engineering for Parallel and Distributed Systems, Kyoto, Japan, April 1998,Page 215-222.

[9] Matthew Dwyer and Lori Clarke; "Data Flow Analysis for verifying properties of concurrent programs"; ACM SIGSOFT'94 Software Engineering Notes, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, P62-75, Dec. 1994.

[10] E. A. Emerson; "Temporal and Model Logic"; In Handbook of Theoretical Computer Science, volume B, chapter 16. Elsevier Science Publishers B. V., 1990.

[11] E. A. Emerson. "Automated Temporal Reasoning about Reactive Systems"; In Logics for Concurrency: Structure versus Automata, LNCS 1043, pages 41-101. Springer-Verlag, 1996.

[12] R. B. France, J. M. Bruel, M. M. Larrondo-Petrie, and M. Shroff; "Exploring the Semantics of UML Type Structures with Z"; IFIP Proc. of Formal Methods in Open Object-Based Distributed Systems, pages 247-257. Chapman & Hall, 1997.

[13] James Gosling, Bill Joy and Guy Steele; "The Language Java Specification"; http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html

[14] K. Havelund and J. U. Skakkebek; "Applying Model Checking in Java Verification"; NASA Ames Research Center, CA, US; http://ase.arc.nasa.gov/havelund; Computer Systems Laboratory, Standford University, Standford, CA, US; http://verify.statndford.edu/jus

[15] K. Havelund and T. Pressburger; "Model Checking Java Programs Using Java PathFinder" ; International Journal on Software Tools for Technology Transfer (STTT) 2(4), April 2000. Special issue containing selected submissions for the 4'th SPIN workshop, Paris, November 1998
http://ase.arc.nasa.gov/havelund;

[16] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept"; Communications of the ACM, 17(10):549, 1974.

[17] Gerard J. Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall, 1991.

[18] Gerard J. Holzmann, "The Model Checker SPIN", IEEE Transactions on Software Engineering, Vol 23, No 5, May 1997.

[19] D. Kozen; "Results on the Propositional μ-calculus"; Theoretical Computer Science, 27(2):333-354, 1983.

[20] S. Leue and P. B. Ladkin; "Implementing and Verifying Scenario-Based Specifications Using Promela/Xspin"; http://www.win.tue.nl/cs/fm/michelr/msc.html

[21] J. Lilius and I. Paltor; "Formalizing UML State Machines for Model Checking"; UML'99: Lecture Notes in Computer Science 1723. Springer-Verlag, 1999.

[22] Jeff Magee and Jeff Kramer; "Concurrency State Models & Java Programs"; John Wiley & Sons Ltd. 1999

[23] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.

[24] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke; "Data Flow Analysis for Checking Properties of Concurrent Java Programs"; Proceedings of the International Conference on Software Engineering (ICSE 1999), p399-410, Los Angeles, CA.

[25] H. Sohn, D. Kung and P. Hsia, Y. Toyoshima, C. Chen; "Reproducible testing for Dsitributed Programs"; 4th Int'l Conference on Telecom. Systems, Modeling and Analysis, Nashiville, Tennessee, P172-179, March, 1996.

[26] J. M. Spivey; "The Z Notation: A Reference Manual"; Prentice Hall, 1992.

[27] C. Stirling; "Modal and Temporal Logics"; In Handbook of Logic in Computer Science, Volumn 2, pages 477-563. Oxford University Press, 1992.

[28] C. Stirling; "Modal and temporal logics for processes"; In Logics for Concurrency: Structure versus Automata, LNCS, 1043, pages 149-237. Springer-Verlag, 1996.

[29] SPIN98 - the 4th International SPIN Workshop
http://netlib.bell-labs.com/netlib/spin/ws98/program.html

[30] Z.100; "ITU Specification and Description Language (SDL)"; Technical Report, ITU, Geneva, 1993.

Appendix

SOURCE CODE FOR TRANSLATOR

```
import java.lang.String;

public synchronized class GlobalVariable {
 // fields
 private String name;
 private String type;
 private String value;
 private String initValue;


 public GlobalVariable(String p0, String p1, String p2) {
  // CONSTRUCTOR implementation not available
 }

 public void setName(String p0) {
  // implementation not available
 }

 public String getName() {
  // implementation not available
 }

 public void setType(String p0) {
  // implementation not available
 }

 public String getType() {
  // implementation not available
 }

 public void setInit(String p0) {
  // implementation not available
 }

 public String getInit() {
  // implementation not available
 }

 public void setValue(String p0) {
  // implementation not available
 }

 public String getValue() {
  // implementation not available
 }

}


public class RCPair
```

```
{
  private String pairName;
  private int numResource;
  private int numCondition;

  public RCPair (String n, int numR, int numC)
  {
        pairName = n;
        numResource = numR;
        numCondition = numC;
  }

  public void setName (String n) {pairName = n;}
  public String getName () {return pairName;}

  public void setNumR (int i) {numResource = i;}
  public int getNumR () {return numResource;}

  public void setNumC (int i) {numCondition = i;}
  public int getNumC () {return numCondition;}

}

import java.util.*;

public class SFSMprocess
{
  private String processName;
  private Vector transitionVector, stateVector, stateNameVec;
  private State initState;

  public SFSMprocess (String n, String init)
  {
        processName = n;
        initState = new State(init);
        transitionVector = new Vector();
        stateVector = new Vector();
        stateNameVec = new Vector();
  }

  public void setInitState (State s0) {initState = s0;}
  public State getInitState () {return initState;}

  public void setProcessName (String n) {processName = n;}
  public String getProcessName () {return processName;}

  public void setTransition (Vector v) {transitionVector = v;}
  public Vector getTransition() {return transitionVector;}

  public void setState (Vector v) {stateVector = v;}
  public Vector getState () {return stateVector;}

  public void setStateName (Vector v) {stateNameVec = v;}
  public Vector getStateName() {return stateNameVec;}

}
```

```
import java.util.*;

public class State
{
  private String stateName;
  private Vector fromTranVec;

  public State (String n)
  {
        stateName = n;
        fromTranVec = new Vector();
  }

  public void setName (String n) {stateName = n;}
  public String getName () {return stateName;}

  public void setVector (Vector v) {fromTranVec = v;}
  public Vector getVector () {return fromTranVec;}

}

class Transition
{
  String startState;
  String endState;
  String guard;
  String action;
  String postAss;

  public Transition (String start, String end, String g, String act, String ass)
  {
        startState = start;
        endState = end;
        guard = g;
        action = act;
        postAss = ass;
  }

  public void setStartState (String s) {startState = s;}
  public String getStartState () {return startState;}

  public void setEndState (String s) {endState = s;}
  public String getEndState () {return endState;}

  public void setGuard (String g) {guard = g;}
  public String getGuard () {return guard;}

  public void setAction (String act) {action = act;}
  public String getAction () {return action;}

  public void setPostAss (String s) {postAss = s;}
  public String getPostAss () {return postAss;}
}

import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class Translator extends JFrame implements ActionListener {
  private JLabel globalLabel, typeLabel, varNameLabel, initLabel, pairsLabel,
          pairNameLabel, numRLabel, numCLabel, processLabel, pNameLabel,
          pInitLabel, tranLabel, startLabel, endLabel, guardLabel, actionLabel,
          postLabel, assertLabel;

  private JTextField typeText, varNameText, initText, pairNameText, numRText,
          numCText, pNameText, pInitText, startText, endText, guardText,
          actionText, postText, assertText;

  private JButton addVar, addPair, addTran, pdone, done, addAssert;

  private JPanel p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13;

  private Vector globalVec, pairVec, pairNameVec, procVec, stateVector, tranVector, assertVec;

  private StringBuffer buf;

  private DataInputStream input;
  private PrintWriter pw;

  public Translator ()
  {
          super ("Translator");

          globalVec = new Vector();
          pairVec = new Vector();
          pairNameVec = new Vector();
          procVec = new Vector();
          stateVector = new Vector();
          tranVector = new Vector();
          assertVec = new Vector();
          buf = new StringBuffer();

          Container c = getContentPane();
          GridLayout gLayout = new GridLayout(13, 1);
          c.setLayout(gLayout);

          p1 = new JPanel();
          p1.setLayout(new FlowLayout());
          globalLabel = new JLabel("Please input info about global variables");
          p1.add(globalLabel);
          c.add(p1);

          p2 = new JPanel();
          p2.setLayout(new GridLayout(1,7));
          typeLabel = new JLabel("varType");
          p2.add(typeLabel);

          typeText = new JTextField(10);
          p2.add(typeText);
```

```
varNameLabel = new JLabel("varName");
p2.add(varNameLabel);

varNameText = new JTextField(15);
p2.add(varNameText);

initLabel = new JLabel("initial value");
p2.add(initLabel);

initText = new JTextField(10);
p2.add(initText);

addVar = new JButton("add variable");
addVar.addActionListener(this);
p2.add (addVar);

c.add(p2);

p3 = new JPanel();
p3.setLayout(new FlowLayout());

pairsLabel = new JLabel("Please input info. about RC_Pairs");
p3.add(pairsLabel);
c.add(p3);

p4 = new JPanel();
p4.setLayout(new GridLayout(1,7));

pairNameLabel = new JLabel("pair name");
p4.add(pairNameLabel);

pairNameText = new JTextField(10);
p4.add(pairNameText);

numRLabel = new JLabel("number of Resources");
p4.add(numRLabel);

numRText = new JTextField(10);
p4.add(numRText);

numCLabel = new JLabel("number of conditions");
p4.add(numCLabel);

numCText = new JTextField(10);
p4.add(numCText);

addPair = new JButton("add RC_Pair");
addPair.addActionListener(this);
p4.add(addPair);
c.add(p4);

p5 = new JPanel();
p5.setLayout(new FlowLayout());

processLabel = new JLabel("Please input info. about your process");
```

```
p5.add(processLabel);
c.add(p5);

p6 = new JPanel();
p6.setLayout(new GridLayout(1,4));

pNameLabel = new JLabel("process name");
p6.add(pNameLabel);

pNameText = new JTextField(10);
p6.add(pNameText);

pInitLabel = new JLabel("initial state");
p6.add(pInitLabel);

pInitText = new JTextField(10);
p6.add(pInitText);
c.add(p6);

p7 = new JPanel();
p7.setLayout(new FlowLayout());

tranLabel = new JLabel("Please input transition info. about current process");
p7.add(tranLabel);
c.add(p7);

p8 = new JPanel();
p8.setLayout(new GridLayout(1,4));

startLabel = new JLabel("start state");
p8.add(startLabel);

startText = new JTextField(10);
p8.add(startText);

endLabel = new JLabel("end state");
p8.add(endLabel);

endText = new JTextField(10);
p8.add(endText);
c.add(p8);

p9 = new JPanel();
p9.setLayout (new FlowLayout());

guardLabel = new JLabel("guard");
p9.add(guardLabel);

guardText = new JTextField(40);
p9.add(guardText);
c.add(p9);

p10 = new JPanel();
p10.setLayout (new FlowLayout());

actionLabel = new JLabel("action");
```

```
        p10.add(actionLabel);

        actionText = new JTextField(40);
        p10.add(actionText);
        c.add(p10);

        p11 = new JPanel();
        p11.setLayout (new FlowLayout());

        postLabel = new JLabel("post assignment");
        p11.add(postLabel);

        postText = new JTextField(35);
        p11.add(postText);
        c.add(p11);

        p12 = new JPanel();
        p12.setLayout(new GridLayout(1,2));

        addTran = new JButton("add Transition");
        addTran.addActionListener(this);
        p12.add(addTran);

        pdone = new JButton("1 process done");
        pdone.addActionListener(this);
        p12.add(pdone);
        c.add(p12);

        p13 = new JPanel();
        p13.setLayout(new GridLayout(1,4));

        assertLabel = new JLabel("please input the invariants");
        p13.add(assertLabel);

        assertText = new JTextField();
        p13.add(assertText);

        addAssert = new JButton("add assertion");
        addAssert.addActionListener(this);
        p13.add(addAssert);

        done = new JButton("input done");
        done.addActionListener(this);
        p13.add(done);
        c.add (p13);

        try {
          input = new DataInputStream(new FileInputStream("promelaProg"));
          pw = new PrintWriter(new FileOutputStream("promelaExt"));
        } catch (Exception ex) { ex.printStackTrace(); }

        setSize (600, 600);
        show();

}
```

```
public void actionPerformed (ActionEvent e)
{
        if (e.getSource() == addVar)
        {
            GlobalVariable gv = new GlobalVariable(typeText.getText(), varNameText.getText(),
initText.getText() );
          globalVec.addElement(gv);
          typeText.setText("");
          varNameText.setText("");
          initText.setText("");
        }
        else if (e.getSource() == addPair)
        {
          RCPair pair = new RCPair (pairNameText.getText(), Integer.parseInt(numRText.getText() ),
Integer.parseInt(numCText.getText()) );
          pairVec.addElement(pair);
          pairNameVec.addElement(pairNameText.getText());
          pairNameText.setText("");
          numRText.setText("");
          numCText.setText("");
        }
        else if (e.getSource() == addTran)
        {
          Transition t = new Transition(startText.getText(), endText.getText(), guardText.getText(),
actionText.getText(), postText.getText() );
          tranVector.addElement(t);
          startText.setText("");
          endText.setText("");
          guardText.setText("");
          actionText.setText("");
          postText.setText("");
        }
        else if (e.getSource() == pdone)
    {
          SFSMprocess proc = new SFSMprocess(pNameText.getText(), pInitText.getText() );
          proc.setTransition(tranVector);

          pNameText.setText("");
          pInitText.setText("");

          Vector vStateName = proc.getStateName();

          Enumeration enum1 = tranVector.elements();


          while (enum1.hasMoreElements() )
          {
                Transition t = (Transition) enum1.nextElement();
                String s1 = t.getStartState();
                String s2 = t.getEndState();

                if (!vStateName.contains(s1) )
                  vStateName.addElement(s1);
                else if (!vStateName.contains(s2) )
                  vStateName.addElement(s2);
```

```
            }

            Enumeration enum3 = vStateName.elements();
            proc.setStateName(vStateName);
            while (enum3.hasMoreElements() )
            {
                    String ss = (String) enum3.nextElement();
          System.out.println("this state is ss");
                    State s = new State(ss);
                    stateVector.addElement(s);
            }

            Enumeration enum2 = stateVector.elements();
            Vector vs = new Vector();

            while (enum2.hasMoreElements() )
            {
                    State s = (State) enum2.nextElement();

                    Enumeration enum4 = tranVector.elements();

                    while (enum4.hasMoreElements() )
                    {
                      Transition t = (Transition) enum4.nextElement();

                      if ( (s.getName()).equals(t.getStartState()) )
                      {
                              Vector v = s.getVector();
                              v.addElement(t);
                              s.setVector(v);
                      }
                    }

                    vs.addElement(s);
            }

            stateVector = vs;
            proc.setState(stateVector);
            procVec.addElement(proc);
            tranVector = new Vector();
          }
          else if (e.getSource() == addAssert)
          {
            assertVec.addElement(assertText.getText());
            assertText.setText("");
          }
          else if (e.getSource() == done)
          {
            translate();
          }
}

public void translate()
{
        String text;
```

```
        try {
 while ( (text = input.readLine() ) != null)
          buf.append(text + "\n");
} catch (Exception ex) {ex.printStackTrace();}

        Enumeration enum1 = globalVec.elements();

        while (enum1.hasMoreElements() )
        {
          GlobalVariable gv = (GlobalVariable) enum1.nextElement();
          buf.append(gv.getType() + " " + gv.getName() + " = " + gv.getInit() + ";\n");
        }

        Enumeration enum2 = procVec.elements();

        while (enum2.hasMoreElements())
        {
          SFSMprocess sfsm = (SFSMprocess) enum2.nextElement();

          buf.append("proctype " + sfsm.getProcessName() + "() { \n");

          Vector vs = sfsm.getState();

          Enumeration enum3 = vs.elements();

          while (enum3.hasMoreElements() )
          {
                State s = (State) enum3.nextElement();

                buf.append(s.getName() + ":  " + "if" + "\n");

                Vector fromTran = s.getVector();

                Enumeration enum4 = fromTran.elements();

                while (enum4.hasMoreElements())
                {
                  Transition tt = (Transition) enum4.nextElement();
                  transitionTranslate(tt);
                  String ss = tt.getEndState();
                  buf.append("goto " + ss + ";\n");
                }
                buf.append("fi\n");
          }
          buf.append("}\n\n\n");
        }

       buf.append("proctype monitor() { assert( \n");

       Enumeration enum7 = assertVec.elements();
       while (enum7.hasMoreElements() )
       {
         String a = (String) enum7.nextElement();
         buf.append(a + ";\n");
       }
```

```
        buf.append(") }\n");
        buf.append("init {\n");
        Enumeration enum5 = pairVec.elements();
        while (enum5.hasMoreElements() )
        {
          RCPair p = (RCPair) enum5.nextElement();
          String s1 = String.valueOf(p.getNumR());
          String s2 = String.valueOf(p.getNumC());
          buf.append("RC_constructor(" + s1 + "," + s2 + ");\n");
        }

        buf.append("atomic {\n");
        Enumeration enum6 = procVec.elements();
        while (enum6.hasMoreElements() )
        {
          SFSMprocess sfsm = (SFSMprocess) enum6.nextElement();
          String name = sfsm.getProcessName();
          buf.append("run "+ name +"();\n");
        }
        buf.append("run monitor(); } }\n");

        try {
  pw.print(buf.toString());
        pw.flush();
 }
 catch (Exception ex) { ex.printStackTrace(); }
 }

 private void transitionTranslate(Transition t)
 {
        String sg = t.getGuard();
        buf.append(":: ");
        while (sg.indexOf(";") != -1)
        {
          String guard = sg.substring(0, sg.indexOf(";")+1);
          sg = sg.substring(sg.indexOf(";")+1);
//        System.out.println("Now guard is " + sg);
          if (guard.indexOf("lock(") != -1)
          {
                int lock1 = guard.indexOf("lock(");
                int lock2 = guard.indexOf(")");
                String pair = guard.substring(lock1+5, lock2);
                int pairIndex = pairNameVec.indexOf(pair);
                buf.append("RC_lock(" + pairIndex + ")\n");
          }
          else buf.append(guard + "\n");
          buf.append("->\n");
/*        if (sg.indexOf(";") == -1)  {
                if(sg.indexOf("lock(") != -1)
                {
                int lock1 = sg.indexOf("lock(");
                int lock2 = sg.indexOf(")");
                String pair = sg.substring(lock1+5, lock2);
                int pairIndex = pairNameVec.indexOf(pair);
                buf.append("lock(" + pairIndex + ")\n");
            }
```

```
                else
                  buf.append(sg + "\n");
            } */
          }

          String act = t.getAction();
          if (act.indexOf("suspend(") != -1 || act.indexOf("resume(") != -1                    ||
act.indexOf("resumeAll(") != -1)
          {
            int p1 = act.indexOf("(");
            int p2 = act.indexOf(",");
            String pair = act.substring(p1+1, p2).trim();
            int pairIndex = pairNameVec.indexOf(pair);
            buf.append(act.substring(0, p1) + "(" + pair +
                  act.substring(p2) + "\n");
          }
          else
            buf.append("printf(\"" + act + "\")\n");

          String post = t.getPostAss();

          while (post.indexOf(";") != -1)
          {
            String postAss = post.substring(0, post.indexOf(";")+1);
            post = post.substring(post.indexOf(";")+1);
            if (postAss.indexOf("unlock(") != -1)
            {
                  int lock1 = postAss.indexOf("lock(");
                  int lock2 = postAss.indexOf(")");
                  String pair = postAss.substring(lock1+5, lock2);
                  int pairIndex = pairNameVec.indexOf(pair);
                  buf.append("RC_unlock(" + pairIndex + ")\n");
            }
            else buf.append(postAss + "\n");
            if (post.indexOf(";") == -1)  {
                  if(post.indexOf("lock(") != -1)
                  {
                  int lock1 = post.indexOf("lock(");
                  int lock2 = post.indexOf(")");
                  String pair = post.substring(lock1+5, lock2);
                  int pairIndex = pairNameVec.indexOf(pair);
                  buf.append("unlock(" + pairIndex + ")\n");
               }
                  else
                    buf.append(post + "\n");
            }
          }
 }


 public static void main(String args[])
 {
        Translator tt = new Translator();

        tt.addWindowListener( new WindowAdapter() {
          public void windowClosing (WindowEvent e)
```

```
            {
                System.exit(0);
            }
        } );
    }

}
```

VITA AUCTORIS

Fang Li was born in 1971 in Zhengzhou, China. She graduated from Tsinghua University, Beijing, P.R.China in 1993, where she received a Bachelor's degree in Civil Engineering. From there she went to work in China Space Civil Engineering Design and Research Institute.

In 1998, she studied in Southwest Texas State University in USA as a speical student. She is currently a candidate for the Master's degree in Computer Science at the University of Windsor and expects to graduate in summer of 2001.